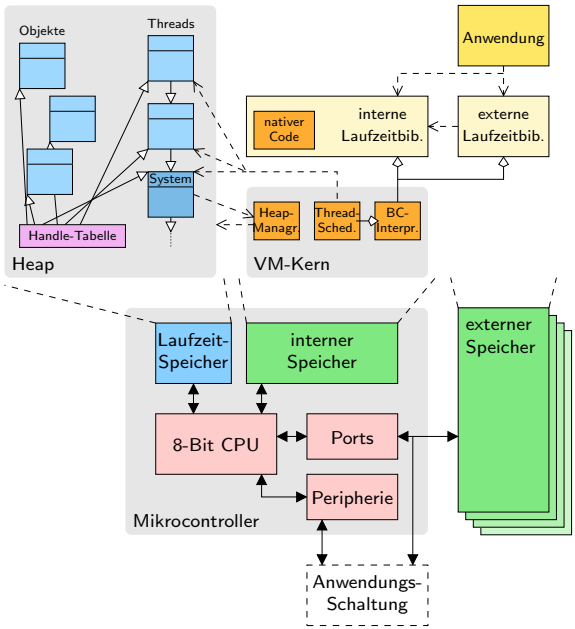
Helge Böhme

Virtuelle Java-Maschinen für kleine eingebettete Systeme



Dissertation, 2007

Abteilung Entwurf integrierter Schaltungen (E. I. S.)
Technische Universität Carolo-Wilhelmina zu Braunschweig
Braunschweig, Deutschland

Helge Böhme

Virtuelle Java-Maschinen für kleine eingebettete Systeme

Dissertation, 2007

Abteilung Entwurf integrierter Schaltungen (E. I. S.)
Technische Universität Carolo-Wilhelmina zu Braunschweig
Braunschweig, Deutschland

TECHNISCHE UNIVERSITÄT
CAROLO-WILHELMINA
ZU BRAUNSCHWEIG

Von der **Carl-Friedrich-Gauß-Fakultät**
für Mathematik und Informatik
der Technischen Universität Braunschweig

genehmigte Dissertation

zur Erlangung des Grades eines
Doktor-Ingenieurs (Dr.-Ing.)

Virtuelle Java-Maschinen für kleine eingebettete Systeme

Helge Böhme

5. Oktober 2006

1. Referent: Prof. Dr. Ulrich Golze

2. Referent: Prof. Dr. Andreas Koch

eingereicht am: 27. Juni 2006

Vorwort

Im Rahmen meines Studiums zum Diplomingenieur der Elektrotechnik nahm ich an Industriepraktika teil. Eines davon wurde indirekt durch die Abteilung Entwurf integrierter Schaltungen (E.I.S.) der Technischen Universität Braunschweig an die mittelständische Firma **aticon**[1] vermittelt. Diese Firma befasst sich mit Home-Automation und entwickelt Techniken für die Vernetzung von Haushaltsgeräten. Dort wurde ich zunächst mit typisch elektrotechnischen Aufgaben betraut (Entwurf und Aufbau von Mikrocontrollerschaltungen), die mich dort auch nach Ende des Praktikums als freier Mitarbeiter ausfüllten. Schließlich wurde mir von Gerrit Telkamp die Möglichkeit gegeben, dort eine studentische Arbeit anzufertigen, wobei mir mehrere Themen zur Wahl standen. Eines davon war, Java auf einem Mikrocontroller zur Ausführung zu bringen und schien die richtige Herausforderung für mich zu sein. Das war 1997 und der Beginn meiner langjährigen Arbeit an diesem Thema.

Rückblickend bewahrheitete sich bei dieser Arbeit eine der inoffiziellen Paradigmen der Informatik bzw. bei der Optimierung von Code:[2]

1. *Make it work!* Dies gelang mir mit der Anfertigung der Diplomarbeit im Jahre 1998. Dieses erste System stellte allerdings eher eine Machbarkeitsstudie dar.

2. *Make it right!* Dies nahm bereits einen Großteil meiner langjährigen Forschungen als wissenschaftlicher Mitarbeiter der Abteilung E.I.S. in Anspruch, welche Ende 1999 begannen. Das resultierende zweite System war bereit für einen produktiven Einsatz.

3. *Make it smart!* Gerade gegen Ende, als das System immer umfangreicher und komplexer wurde, zeigte sich ein hoher Aufwand bei Fehlerbereinigungen und Erweiterungen. Ich begann also 2004 über komplett neue Entwurfsmethoden nachzudenken.

Diese neuen Entwurfsmethoden finden sich am Ende der Dissertationsschrift wieder, welche zwar den Abschluss meiner Forschungen an der Abteilung E.I.S.

[1] Diese Firma ging später in `domo:logic` über.

[2] Frei zitiert nach Kent Beck [Bec99], dem die erste Nennung der drei Sätze „*Make it work. – Make it right. – Make it fast.*“ nachgesagt wird.

darstellt aber vermutlich nicht das Ende meiner Arbeit an und mit Java für eingebettete Systeme.

Danksagung

Die vorliegende Arbeit wäre ohne Hilfe nicht zustande gekommen. Besonderer Dank gilt meinem Betreuer und Mentor, Herrn Prof. Dr. Ulrich Golze, der mich mit beinahe grenzenloser Geduld bei meinen Forschungen und der Anfertigung dieser Arbeit begleitet hat. Nicht weniger verdanke ich Herrn Gerrit Telkamp als Projektinitiator und -partner. Ich erinnere mich noch lebhaft an unsere zahlreichen Diskussionen der großen Konzepte und der kleinen Vorgehensweisen bei unseren gemeinsamen Forschungen und Entwicklungen. Danken möchte ich auch den an dieser Stelle nicht im einzelnen namentlich genannten Studenten der TU Braunschweig und den Mitarbeitern der Firma `domo:logic`, die Beiträge zu meinen Forschungen erarbeiteten. Dies gilt auch für die Kollegen der Abteilung E.I.S., die nicht nur Diskussionsbeiträge beisteuerten, sondern auch ein angenehmes kollegiales Umfeld geschaffen haben. Schließlich gilt mein Dank meiner Familie, die meine Ausbildung ermöglicht und mein Wirken unterstützt hat.

Kurzfassung

Die objektorientierte Programmiersprache Java ist auf eingebetteten Systemen noch nicht stark verbreitet, meist aus Kostengründen. Eine virtuelle Java Maschine erfordert normalerweise ein leistungsfähigeres System. Diese Arbeit befasst sich damit, Java auf besonders preiswerten 8-Bit-Mikrocontrollern auszuführen. Das eröffnet für Java die Welt der Messung, Steuerung und Regelung und verknüpft sie mit Benutzerinteraktion und Kommunikation. Java kann mit geringerem Programmieraufwand dazu beitragen, beispielsweise einen Haushalt zu steuern und zu überwachen.

Für die speichereffiziente Umsetzung von Java werden einige Techniken evaluiert und auf einem Mikrocontroller (ST7) integriert. Mittels einer Vorverarbeitung von Java-Programmen und der virtuellen Maschine selbst auf einem Entwicklungssystem wird der Platzbedarf auf dem Zielsystem verringert. Geeignete Datenstrukturen und Klassenbibliotheken (API) belegen nur wenig Laufzeitspeicher mit Daten. Die Kombination von Java-Bytecode mit zielsystemabhängigen nativen Code ermöglicht die Ansteuerung von Peripheriekomponenten. Geeignete Zeitsteuerungen (zeitschrankenbasiertes Thread-Scheduling) machen Java auch für zeitkritische Anwendungen geeignet. Zusammen mit einer Software-Umgebung auf dem Entwicklungssystem entstand ein einsatzfähiges Java-System für einen Mikrocontroller.

Die bei der Entwicklung dieser JavaVM gemachten Erfahrungen werden schließlich in neue Entwurfsverfahren zur Erstellung spezieller eingebetteter virtueller Maschinen umgesetzt. Dabei kommt ein vollständig in Java beschriebenes mehrschichtiges Modell zum Einsatz, das sich flexibel an verschiedene Zielsysteme anpassen lässt. Das Modell enthält die virtuelle Maschine bestehend aus Kern und Laufzeitbibliothek sowie eine allgemeine Anwendungsprogrammierschnittstelle. Erst Code-Generatoren fügen den zielsystemabhängigen nativen Code in das Modell ein.

Abstract

Java is an object oriented programming language. But mainly because of the costs it's not widely used on embedded systems. Typical Java virtual machines require larger systems. This work aims for integration of Java on inexpensive 8-bit microcontrollers. This makes Java possible in the world of measurement, control and automation and allows the combination of control, user interaction and communication on a single system. E. g. Java can be used to enable home automation with less programming effort.

To integrate Java on small embedded systems with low memory consumption, some techniques are explored and realized on a microcontroller (ST7). Due to preprocessing of Java programs and the virtual machine itself on a development system, the program memory allocation on the target system is reduced. Applicable data structures and class libraries (API) are designed to use as little data memory as possible. If Java bytecode is combined with the target system's native code, peripheral components can be utilized. Furthermore deadline based thread scheduling can be used for time critical tasks. Together with a software environment for development systems this results in an operative Java system on a microcontroller.

Experiences developing this JavaVM are now resulting to new development practices for creation of specialized embedded virtual machines. A completely Java based multilayered model is able to match various target systems. The model contains the virtual machine (kernel and runtime) and a generic application programming interface. Only at the end, code generators insert target system specific native code into the model.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

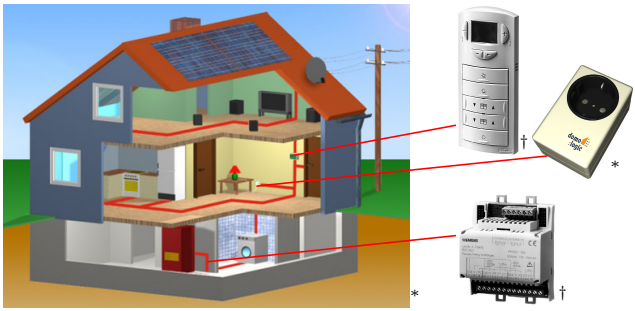
Kapitel 1

Einleitung

Diese Arbeit beschäftigt sich damit, die Programmiersprache Java auf Mikrocontrollern, also eingebetteten Systemen, auszuführen. Da Java auf dem PC- und Server-Sektor eher für seinen Ressourcenhunger bekannt ist, stellen sich Fragen nach Praktikabilität und Einsatzzweck. Um dies zu erläutern, soll an dieser Stelle ein Blick auf die Ursprünge des Projekts geworfen und die eigentliche Motivation dargestellt werden.

1.1 Motivation: Home-Automation

Die Forschungsgruppe Home-Automation der Abteilung Entwurf integrierter Schaltungen (E.I.S.) der Technischen Universität Braunschweig kooperiert mit der ortsansässigen mittelständischen Firma `domo:logic` Home-Automation GmbH. Diese befasst sich mit kostengünstigen Techniken und Verfahren zur Vernetzung von Haushaltsgeräten. Bild 1.1 skizziert die Vision des zukünftigen



*© Marcus Timmermann, `domo:logic` Home-Automation GmbH
†© Landis & Staefa GmbH / Siemens Building Technologies GmbH

Bild 1.1: Der vernetzte Haushalt

Haushalts. Alle Geräte sind irgendwie miteinander verbunden und integrieren sich in ein funktionierendes Haus-System. Diesem System können nun vielerlei Aufgaben übertragen werden. Dies geht von einfachen Komfortfunktionen (z. B. das Abrufen vorprogrammierter Lichtstimmungen im Wohnzimmer) über Energiemanagement (Heizungssteuerung in Abhängigkeit vom Aufenthaltsort der Bewohner und Zustand der Fenster) bis zu Sicherheitsfunktionen (beim Verlassen des Hauses werden „vergessene“ Geräte abgeschaltet).

Für die Vernetzung der Geräte ist ein Bus-System nötig. Es deckt dabei weniger den Datenaustausch zwischen einem PC und diversen HiFi- und TV-Geräten ab, sondern eher zwischen Lichtschaltern und Lampen, also kleinen Komponenten, von denen in einem Haushalt viele existieren. Diese Technik hat sich heutzutage daher meist nur bei Bürogebäuden durchsetzen können, für den Privathaushalt ist sie noch zu kostspielig und aufwändig bei Installation und Betrieb. Ziel ist es also, die Kosten und den Installationsaufwand für das Netzwerk und die Netzwerk-Knoten zu senken und die Bedienung zu vereinfachen.

1.1.1 Der Haushalt als Feldbus

Um verschiedenen Einsatzprofilen gerecht zu werden, haben sich unterschiedliche Bus-Systeme etablieren können. In Neubauten sind das preiswerte drahtgebundene Systeme ähnlich den Feldbus-Systemen im industriellen (ProFiBus) oder automotiven (Conrol Area Network; CAN) Umfeld. In Altgebäuden mit bereits vorhandener elektrischer Installation kommen hingegen eher Konzepte basierend auf Funk- oder Power-Line-Kommunikation (also direkt über die Stromleitung) zum Einsatz. Als Kommunikationsstandards haben sich im Wesentlichen KNX (Konnex; ein Zusammenschluss von EIB, EHS und BatiBUS) [1] und LON (Local Operating Network) [2] durchgesetzt, wobei von einigen Standards mehrere Kommunikationsmedien unterstützt werden. Je nach den jeweiligen Voraussetzungen können also Haushaltsvernetzungen basiered auf unterschiedlichen Medien, Topologien oder Protokollen eingerichtet werden, auch hybride Lösungen sind möglich.

Zur Anbindung eines Netzwerk-Knotens an einen solchen Bus ist eine Software-Komponente erforderlich, die ein Kommunikationsprotokoll realisiert. Dabei wird in den vorhandenen Standards sich an dem ISO/OSI-Referenzmodell orientiert. In der Home-Automation werden dann innerhalb dieses Protokolls Geräte- und Dienstenummern sowie Datentypen definiert, die dann im Netzwerk sichtbar sind. Bei der Netzwerk-Installation müssen diese Nummern den Steuergeräten bekannt gemacht werden (der Lichtschalter muss die Adresse der

Lampe kennen). Eine Umkonfigurierung im laufenden Betrieb eines solchen Netzwerks funktioniert dann meist von zentraler Stelle aus.

1.1.2 Die Mensch-Maschine Schnittstelle

Eine weitere Aufgabe kommt diesen Netzwerk-Knoten noch zu: Sie stellen eine Schnittstelle zwischen den Geräten und den Bewohnern des Hauses dar. Auf der einen Seite findet eine Steuerung der Gerätefunktionen statt. Der Mikrocontroller übernimmt dabei die Aufgaben, die bisher mechanische Komponenten (Schalter) erfüllt haben. Dabei wird ggf. auf Messwerte von Sensoren zurück gegriffen, so dass weniger Aufmerksamkeit des Benutzers für den Betrieb des Geräts notwendig ist. Viele Geräte werden heute bereits von eingebetteten Systemen gesteuert (von der Mikrowelle bis zum Toaster mit eingebautem Bräunungssensor), hier muss eigentlich nur die Netzwerk-Komponente integriert werden. Häufig ist dies allerdings mit einem kompletten Neuentwurf der Gerätesteuerung gleichzusetzten, werden diese doch auch kostenorientiert und nicht erweiterungsorientiert produziert.

Auf der anderen Seite steht der Mensch, der diesen vernetzten Haushalt bedienen will. Hierfür sollen also einfache und intuitive Konzepte vorhanden sein, der klassische Lichtschalter soll also nicht durch etwas Komplizierteres ersetzt werden, sondern allenfalls ergänzt. Oft wird dazu in den Medien die Steuerung des Haushalts über den Fernseher präsentiert. Diese Lösung ist sicherlich im Wohnzimmer optimal, nicht aber in der Küche oder gar im Keller. Stattdessen ist es möglich, im Haus Bedienkonsolen zu verteilen, die über eine einfache grafische Menüsteuerung den Haushalt konfigurieren und überwachen können.

Hierbei spielen nicht nur intuitive Bedienkonzepte, sondern auch die Gesamtkosten des vernetzten Heims eine Rolle, so dürfen die Kosten dieser vernetzten Installation die einer gewöhnlichen nicht vernetzten nicht zu stark überschreiten, um für private Haushalte interessant zu sein. Die Kosten je Knoten teilen sich dabei auf in die Hardware-Kosten (es sollen möglichst wenige und preiswerte Komponenten verwendet werden) und die Entwicklungs- und Einrichtungskosten. Hierbei zeigt sich das Problem beim vernetzten Heim, denn diese Netzwerke sind keinesfalls gleichförmig und die einzelnen Knoten müssen u. U. individuell konfiguriert oder gar programmiert und eingerichtet werden. Hierfür sollen modernere Konzepte, als die bei Mikrocontrollern meist noch übliche Programmierung in maschinennahen Sprachen wie **C** oder Assembler, eingesetzt werden.

1.2 Java in der Home-Automation

Die objektorientierte Programmiersprache Java bietet sich an, um das Problem der aufwändigen Programmierung und Konfigurierung der Netzwerk-Knoten abzumildern. Gerade auf eingebetteten Systemen spielt Java mit dem plattformübergreifenden Konzept der virtuellen Maschine seine Stärken aus. Geräte verschiedener Hersteller und mit unterschiedlichen Hardware-Konzepten können dann nicht nur in ein gemeinsames Netzwerk integriert werden, weil sie dasselbe Netzwerk-Protokoll sprechen, sondern es sind auch die verwendeten Software-Komponenten untereinander kompatibel und sogar austauschbar.

1.2.1 Die Java-Geschichte

Besonders interessant ist in diesem Fall die Geschichte von Java. Sie geht auf das Jahr 1991 zurück, als bei *Sun Microsystems* das „Green Team“ die Programmiersprache Oak entwickelte [3], sie wurde später in Java umbenannt. Einer der Leiter des Projekts, James Gosling, galt später als der Vater von Java. Ziel der Forschungen des „Green Projects“ war es, die Welt der Computer mit der des Haushalts zu vereinigen. Herausgekommen ist als Demonstration das *7 (Star-Seven) Gerät, eine Art vergrößerte Fernbedienung mit berührungsempfindlicher grafischer Anzeige. Der Einsatz von Java auf eingebetteten Systemen zur Steuerung von Haushaltsgeräten geht also auf seine Ursprünge zurück.

1.2.2 Java als Programmiersprache für Netzwerk-Knoten

Auf den Netzwerk-Knoten im Speziellen bietet Java (neben weiteren Vorteilen auf kleinen eingebetteten Systemen im Allgemeinen, die im Rahmen dieser Arbeit ebenfalls erörtert werden) insbesondere zwei herausragende Merkmale, die mit anderen, maschinennäheren Programmiersprachen nur schwierig umsetzbar sind:

1. Die Plattformunabhängigkeit der virtuellen Java Maschine ermöglicht den individuellen Austausch von Programmen auf den Netzwerk-Knoten. Unter der Voraussetzung einer gemeinsamen Programmierschnittstelle (API) auf allen Geräten können die Hersteller die passenden Bedienelemente für im Netzwerk vorhandene Steuerkonsolen (auch anderer Hersteller) mitliefern.
2. Die Objektorientierung der Programmiersprache Java und deren direkte Unterstützung durch die virtuelle Maschine ermöglicht es, die abstrakten Dienste und Datentypen der Netzwerk-Protokolle durch selbsterklärende

Software-Komponenten zu ergänzen. Unter der Voraussetzung einer gemeinsamen Programmierschnittstelle wird die Programmierung der Netzwerk-Knoten vereinfacht und standardisiert.

Beide Punkte zusammen ermöglichen aus Programmierersicht, die realen Gegenstände eines Haushalts auf entsprechende Java-Objekte abzubilden. Jedes Gerät verfügt über einen individuellen Satz von Parametern und Aktionen, die sowohl lokal auf dem Gerät vorhanden sind (und dort direkt wirken) als auch auf den Steuerkonsolen (und dort eine Kommunikation über das Netzwerk bewirken). Java ermöglicht es also, das Netzwerk vor dem Programmierer zu verbergen. Wie oben schon angedeutet, spielt dabei eine gemeinsame Programmierschnittstelle (API) die entscheidende Rolle. Dieses API dient dazu, die maschinennahen Komponenten, die auf einem Mikrocontroller existieren, mit den abstrakten Komponenten des Haushalts zu verbinden; sinnvoll ist dabei eine Aufteilung in mehrere Schichten:

Hardware-Ebene Es werden einfachste Ein- und Ausgabefunktionen bedient, z. B. das Setzen oder Auslesen von Steuerleitungen und -bussen, aber auch automatisch ablaufende einfache Prozesse, z. B. die Generierung von pulsweitenmodulierten Signalen (PWM) oder Zugriffe auf dedizierten Datenspeicher.

Komponentenorientierte Ebene Hier sind Zugriffe auf direkt mit dem Gerät verbundene Peripheriekomponenten zusammengefasst, z. B. ein Display, eine Tastatur oder die Komponenten des zu steuernden Haushaltsgeräts, also Motoren, Lampen etc. Ferner findet sich hier die Schnittstelle zum Haushalts-Netzwerk wieder.

Geräteorientierte Ebene In der obersten Schicht ist nun die gewünschte Abstraktion bei der Gerätesteuerung vorhanden. Hier existieren nur noch komplexe Komponenten, also komplette Haushaltsgeräte oder Steuerkonsolen. Dies kann das eigene Gerät sein (auf dem sich der Mikrocontroller befindet, auf dem das Programm ausgeführt wird) oder ein entferntes (und über das Netzwerk ferngesteuertes oder -überwachtes).

Bild 1.2 skizziert dieses Szenario anhand des Beispiels einer Kaffeemaschine. Auf beiden miteinander vernetzten Geräten befindet sich eine JavaVM, die jeweils über die gleiche Java-Programmierschnittstelle verfügt. Lediglich Unterschiede vorhandener Hardwarekomponenten spiegeln sich als unterschiedliche Software-Varianten wider, das betrifft aber nur die An- oder Abwesenheit von Software-Modulen (Display, Tastatur, Heizplatte), die Schnittstellen der Module mit den gleichen Aufgaben bleiben identisch. Die Kaffeemaschine stellt nun zwei Varianten der eigenen Gerätesteuerungskomponente bereit, eine lokal funktionierende, die ggf. Kommandos vom Netzwerk entgegennimmt, und eine

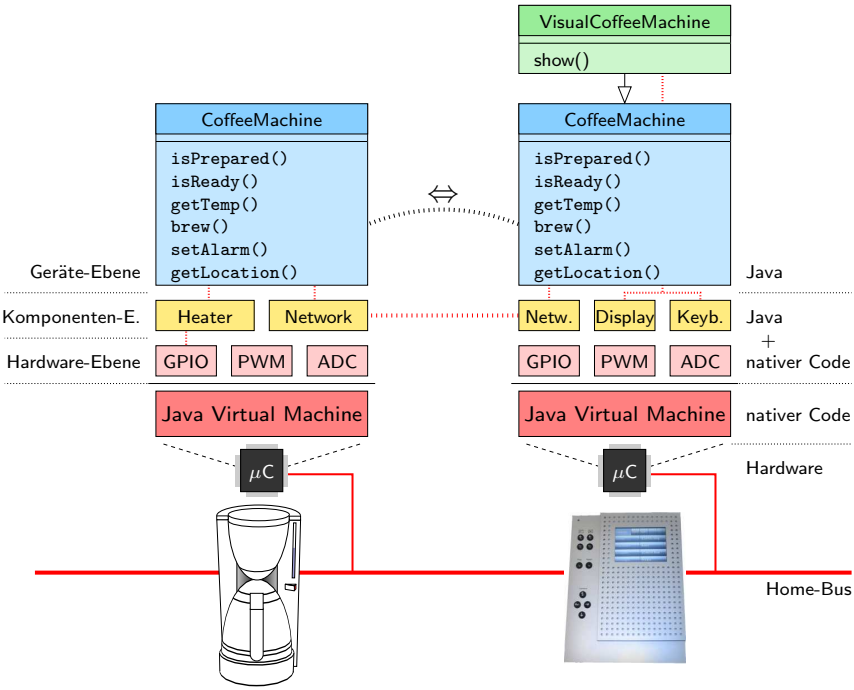


Bild 1.2: Home-Automation mit Java

für die Übertragung auf Steuerkonsolen bestimmte, welche ausschließlich Kommandos über das Netzwerk verschickt. Wichtig ist, dass beide Ausprägungen dieselbe Schnittstelle bedienen (nämlich die für Kaffeemaschinen); das gilt dann für alle Kaffeemaschinen, auch anderer Hersteller. Eine Bedienkonsole kann sich nun immer auf ein solches Modul beziehen, wenn eine grafische Benutzerschnittstelle für eine Kaffeemaschine dargestellt werden soll. Ähnliches gilt für die Netzwerkschnittstelle, welche unabhängig von den verwendeten Bus-Systemen, Protokollen oder Datentypen immer dieselbe Programmierschnittstelle zur Verfügung stellt.

Ziel dieser Arbeit ist, die entsprechenden Voraussetzungen für eine solche Plattform in der Home-Automation zu schaffen, eine kostengünstige virtuelle Java-Maschine für kleine eingebettete Systeme.

1.3 Gliederung und Nomenklatur

Diese Arbeit gliedert sich wie folgt. Zunächst werden das Konzept und die Realisierung einer speziellen virtuellen Java-Maschine auf einem Mikrocontroller beschrieben, die Yogi2-VM. Nach einer Einführung in die Grundlagen und einem Vergleich mit anderen Realisierungen wird der grundlegende Aufbau dieser Realisierung dargestellt. Die anschließenden Kapitel 5 bis 7 beleuchten die wesentlichen Kernkomponenten der JavaVM im einzelnen: die Speicherverwaltung, den Bytecode-Interpreter und den Thread-Scheduler. Abgeschlossen wird die Yogi2-VM mit einem Blick auf die Laufzeitumgebung und spezielle Erweiterungen.

Nach einer Zwischenbilanz wird in Kapitel 10 ein Ausblick auf neue Konzepte zur Realisierung künftiger virtueller Java Maschinen auf eingebetteten Systemen gegeben. Dabei sollen moderne und teils neue Softwaretechniken eingesetzt werden. Eine Zusammenfassung schließt diese Arbeit ab.

In dieser Arbeit werden verschiedene Schriften verwendet, um den Text besser lesbar zu machen. In **serifenloser Fettschrift** erscheinen Kapitelüberschriften und Aufzählungsrubriken. Im Fließtext wird die erste wichtige Nennung von Schlüsselworten mit *Kursivschrift* hervorgehoben, diese Begriffe erscheinen im Index am Ende der Arbeit zusammen mit den Seiten erneuten Auftretens. Herausragende sprachliche Betonungen erscheinen ebenfalls *kursiv*. Quelltext-Ausschnitte und Zitate aus ihnen im Fließtext werden in der Schriftart `Typewriter` gesetzt, so dass sie eindeutig erkennbar sind. Firmennamen erscheinen in einer Schriftart, die der des offiziellen Logos am ähnlichsten ist.

Literaturhinweise werden in eckigen Klammern notiert [Lor97]. Dabei wird ein Kürzel aus Autorennamen und Erscheinungsjahr verwendet; dieses Kürzel ist die Grundlage für die Sortierung des Literaturverzeichnisses. Verweise auf Internetquellen haben naturgemäß kein Erscheinungsjahr und selten einen eindeutigen Autor, sie werden daher der Reihe nach durchnummeriert [0] und separat gelistet. Die Liste kann natürlich nur eine Momentaufnahme des Internets zum Zeitpunkt der Anfertigung dieser Arbeit bieten. Sofern möglich, werden daher unter [Böh06c] archivierte Fassungen der Quellen zur Verfügung gestellt.

Kapitel 2

Grundlagen

Bevor auf die speziellen Eigenschaften der hier vorgestellten virtuellen Java-Maschine für kleine eingebettete Systeme eingegangen wird, sollen zunächst einige damit zusammenhängende Begriffe erklärt und ein Überblick über die Java-Technologie gegeben werden.

2.1 Komponenten einer JavaVM

Die virtuelle Java-Maschine ist – wie ihr Name schon andeutet – ein in Software nachgebildeter Prozessor. Sie verfügt über einen eigenen Befehlssatz, den Java-Bytecode. Dieser ist auf allen Implementierungen gleich, so dass Java-Anwendungen auf allen Systemen binärkompatibel sind. Der Aufbau und die Funktionen der JavaVM wurden von *Sun Microsystems* in einer *Spezifikation* [LY00] festgehalten, die Beschreibungen in dieser Spezifikation reichen aus, um eine virtuelle Java-Maschine zu implementieren.[3]

Der Java-Quelltext wird von einem Compiler in die kleinsten Einheiten einer Java-Anwendung, die Klassendateien (Class-Files) übersetzt. Wie diese Übersetzung erfolgt, wurde ebenfalls von *Sun* in einer *Sprachspezifikation* [BGJS00] festgeschrieben. Bild 2.1 zeigt den Prozess vom Quelltext zur Ausführung einer Anwendung auf einem Entwicklungssystem. Heutzutage wird üblicherweise statt des einfachen Java-Compilers von *Sun*, der im Java Development Kit (JDK) zur Verfügung steht und kommandozeilenorientiert arbeitet, eine *Integrierte Entwicklungsumgebung* (IDE) verwendet, welche sich positiv auf den Entwicklungsprozess auswirkt. So finden sich dort viele Werkzeuge, die die Programmerstellung unterstützen und beschleunigen; sie sollen an dieser Stelle nur kurz als Stichworte genannt werden: Inhaltsassistent (Content-Assist), Elementeverzeichnis (Class-Outline), Projektverwaltung, Versionsmanagement

[3] Wird *nur* die Spezifikation für eine Implementierung herangezogen, so handelt es sich um eine sogenannte Clean-Room-Implementierung, also ohne Einflüsse von außen durch vorhandenen bzw. fremden Programm-Code.

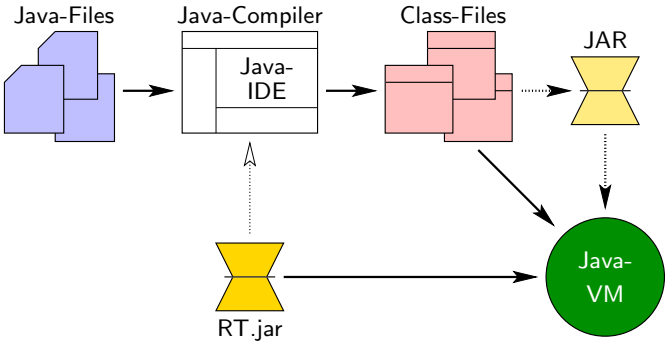


Bild 2.1: Der Entwicklungsfluss von Java-Anwendungen

(z. B. CVS), Debugger. Die Klassendateien können nach Bedarf auch zu einem Java-Archiv (*JAR*) zusammengefasst werden, was Vorteile beim Vertrieb und Start von Anwendungen bedeutet.

Jede der übersetzten Klassen verfügt über einen Satz von Attributen (Konstanten, Variablen) und Methoden (sie enthalten den in Bytecode übersetzten Programmcode) und spiegelt die objektorientierte Natur von Java wider. Die Objektorientierung ist also auf der Seite der virtuellen Maschine immer noch sichtbar und kann dort ausgenutzt werden. Dies unterscheidet Java grundsätzlich von anderen objektorientierten Programmiersprachen, die für einen realen Prozessor übersetzt werden. Dort wird die Objektorientierung vom Compiler aufgelöst. Bei Java findet also innerhalb des Entwicklungsflusses eine späte Auflösung (Late Binding) der Bezüge (Referenzen) zwischen den einzelnen Klassen statt: beim Start einer Anwendung bzw. beim ersten Zugriff auf eine Klassendatei bei der Ausführung einer Anwendung. Dies hat den großen Vorteil, dass nicht alle Programmteile bei der Übersetzung vorhanden sein müssen und auch bei einem bereites laufenden System später weitere Klassen hinzugefügt werden können.

Java wird aber nicht nur durch eine standardisierte Programmiersprache und virtuelle Maschine bestimmt. Eine (mittlerweile sehr umfangreiche) Laufzeitbibliothek (Programmierschnittstelle; API) macht einen Großteil der für einen Programmierer verfügbaren Funktionalität aus. Die Laufzeitbibliothek ist in Bild 2.1 als RT.jar zu erkennen. Sie wird einerseits vom Compiler (bzw. der Entwicklungsumgebung) zum Übersetzen benötigt, um die Bezüge einer Anwendung zu verifizieren, und andererseits von der ausführenden virtuellen Maschine. Die Laufzeitbibliothek enthält nichts als weitere Klassendateien mit

Daten und Code, die von der Anwendung mitbenutzt werden. Die virtuelle Maschine und die Laufzeitbibliothek werden zu der Java-Laufzeitumgebung (*Java Runtime Environment*; JRE) zusammengefasst. Von *Sun* werden mittlerweile drei wesentliche Varianten dieser Laufzeitumgebung zur Verfügung gestellt, die sich in Umfang und Funktionalität unterscheiden [4]. Die Java 2 Standard Edition (J2SE) ist für den Einsatz auf Arbeitsplatzsystemen ausgelegt und unterstützt beispielsweise grafische Bedienoberflächen. Die Java 2 Enterprise Edition (J2EE) ist für den Betrieb auf Firmen- und Netzwerk-Servern gedacht und bietet u. a. Datenbank-Anbindungen und Application-Server. Die Java 2 Micro Edition (J2ME) ist für den Einsatz auf eingebetteten Systemen vorgesehen.

Teil aller Laufzeitumgebungen sind spezielle Mechanismen, die die Sprache Java aus Programmierersicht aufwerten. Dies ist zum einen die Speicherverwaltung, die sich selbst vor dem Programmierer verbirgt (Java kennt keine Adressen) und dadurch mögliche Fehlerquellen bei der Programmentwicklung eliminiert. Teil der Speicherverwaltung ist eine automatische Bereinigung von nicht mehr benötigten Objekten (Instanzen und Laufzeitrepräsentationen von Klassen im Speicher), die Garbage-Collection. Ein weiteres herausragendes Merkmal ist die eingebaute Fähigkeit, mehrfädige Anwendungen abzuarbeiten (Multi-Threading). Dabei werden parallele Abläufe gleichzeitig auf mehreren Prozessoren oder quasi gleichzeitig (im schnellen Wechsel) auf einem Prozessor abgewickelt. Diese Mehrfädigkeit macht Java insbesondere auf eingebetteten Systemen interessant, vereinfacht sie doch die Entwicklung von reaktiven Anwendungen, die schnell auf externe Ereignisse (z. B. von Sensoren) reagieren müssen. In diesem Bereich finden immer noch viele Forschungen statt, um dieses *Echtzeit*verhalten, insbesondere im Zusammenhang mit der Speicherbereinigung, zu verbessern.

Schließlich existieren noch zahlreiche Optimierungen der virtuellen Maschine, meist um die Ausführungsgeschwindigkeit zu verbessern. Die einfachste Möglichkeit, Java-Anwendungen auf einem realen Prozessor mit einem eigenen Befehlssatz auszuführen, ist die Bytecode-Interpretierung. Dabei wird der Bytecode programmgesteuert eingelesen und anweisungsweise ausgewertet. Dieses Verfahren ist zwar mit wenig Aufwand umsetzbar und somit insbesondere für eingebettete Systeme geeignet aber auch entsprechend langsam in der Ausführung. Auf größeren Systemen haben sich daher die Laufzeitcompiler (Just-In-Time; JIT) durchgesetzt, die beim Laden und Auflösen einer Klasse den Bytecode in für den Zielprozessor bestimmten nativen Code übersetzen. Da dieser Übersetzungsvorgang seinerseits Zeit und Speicherplatz kostet, gibt es auch hybride Lösungen, die nur einen zeitkritischen Teil der Anwendung in nativen Code übersetzen. Dazu wird eine Analyse des Laufzeitverhaltens (Profiling)

durchgeführt.[4]

2.2 Java für eingebettete Systeme

Für eingebettete Systeme unterscheidet sich der Entwicklungsfluss geringfügig gegenüber dem für Arbeitsplatzsysteme (wie in Bild 2.2 gezeigt). Bis zu ei-

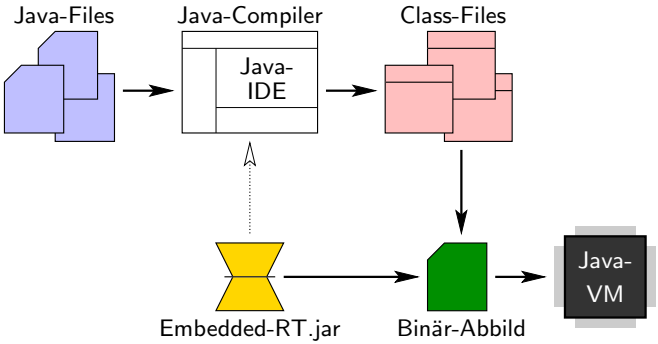


Bild 2.2: Der Entwicklungsfluss von eingebetteten Java-Anwendungen

ner bestimmten Größe des Zielsystems wird auf die Fähigkeit verzichtet, zur Laufzeit von Anwendungen Klassendateien hinzuzufügen. Hierbei werden die Referenzen zwischen den Klassen *vor* der Ausführung der Anwendung auf dem eingebetteten System bereits auf dem Entwicklungssystem aufgelöst und ein Binär-Abbild der Klassen erzeugt (Early Binding).[5] Das liegt hauptsächlich darin begründet, dass die Auflösung der Referenzen ein speicherintensiver Vorgang ist. Resultat sind Sprungtabellen für den Aufruf von Methoden und Referenztabellen für den Zugriff auf Variablen. Kleine eingebettete Systeme, insbesondere Ein-Chip-Lösungen auf einem Mikrocontroller, verfügen nur über sehr wenig Laufzeitspeicher (RAM), der von diesen Tabellen schnell gefüllt würde. Da diese Tabellen nur einmalig erzeugt werden müssen und sich zur Laufzeit nicht verändern, ist es sinnvoller, sie im Festwertspeicher (ROM) des Controllers zusammen mit dem Bytecode abzulegen. Als weiterer Effekt kann auf die

[4] Bekanntestes Beispiel dafür ist der Hotspot-Compiler von Sun, welcher bei der J2SE und J2EE eingesetzt wird [5].

[5] Dieser Prozess wird bei vielen Realisierungen als ROMizing bezeichnet, der Begriff wird z. B. bei der KVM (Kilobyte Virtual Machine) von *Sun* verwendet, welche die J2ME realisiert, siehe Abschnitt 3.1.1.1.

Klassen, auf die kein Bezug durch die Anwendung existiert, auf dem Zielsystem verzichtet werden. Je nach Zielsystem enthält dieses Binär-Abbild nur die Klassen oder zusätzlich noch die virtuelle Maschine selbst (zusammen mit dem übrigen nativen Code).

Auf eingebetteten Systemen werden verschiedene Optimierungen eingesetzt, um beispielsweise den benötigten Speicherplatz zu reduzieren, oder die Ausführungsgeschwindigkeit zu verbessern. Letzteres wird erreicht, indem der Java-Bytecode auf dem Entwicklungssystem in nativen Code für das Zielsystem übersetzt wird und mit dem Binär-Abbild übertragen wird (Ahead-Of-Time Compilierung). Je nach Zielsystem wird dabei allerdings auch mehr Festwertspeicherplatz belegt.

Das für den Anwendungsentwickler sichtbare hauptsächliche Unterscheidungskriterium gegenüber Java für Arbeitsplatzsysteme ist jedoch die verfügbare Laufzeitbibliothek. Diese ist stark an Zielsystem und Einsatzgebiet orientiert. Auffällig ist eine stärkere Koppelung an die mit dem Zielsystem verwendete Hardware, welche bei der Realisierung von *Sun*, der J2ME, nur sehr marginal in Erscheinung tritt (hier müssen sich die Zielsysteme an die Java-Laufzeitbibliothek anpassen, somit werden die Einsatzmöglichkeiten beschnitten). Bei noch kleineren Systemen, deren Aufgabe insbesondere darin besteht, Geräte zu steuern, ist dies aber entsprechend auffälliger. Hierbei ist durch den Entwurf der Laufzeitbibliothek ein guter Kompromiss zwischen Portabilität und Spezialisierung zu schaffen.

Kapitel 3

Vergleich verschiedener Implementierungen

Das hier vorgestellte Konzept einer virtuellen Java-Maschine auf einem 8-Bit-Mikrocontroller entstand zu einer Zeit, als eine sinnvolle Verwendung von Java auf eingebetteten Systemen wenig diskutiert wurde. Im Jahre 1998 existierte die J2ME nur in ihren Grundzügen (damals noch unter dem Namen EmbeddedJava und PersonalJava) und als weiteres Projekt die Java Card. Bild 3.1 zeigt einen Überblick über diese Varianten mit einer Abschätzung der jeweili-

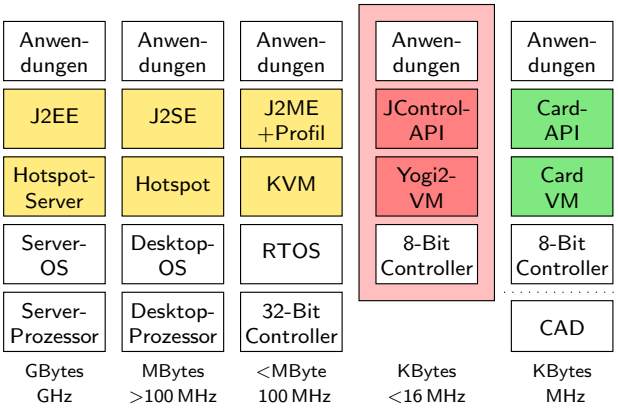


Bild 3.1: Vergleich von Java-Varianten

gen Systemanforderungen. Die hier realisierte Implementierung der Yogi2-VM liegt dabei von ihren Anforderungen zwischen der Java Card und der J2ME. Im Folgenden sollen einige Systeme in diesem Bereich kurz vorgestellt werden, sie entstanden meist zeitgleich mit diesem Projekt. Eine umfangreichere Übersicht über vorhandene JavaVMs findet sich z. B. in [6].

3.1 Virtuelle Maschinen für eingebettete 32-Bit-Systeme

Meist werden 32-Bit-Systeme für die Ausführung von Java-Anwendungen vorausgesetzt, von denen hier einige vorgestellt werden sollen.

3.1.1 J2ME

Die Java 2 Micro Edition [7, Top02] spezifiziert weniger eine VM-Architektur, sondern im Wesentlichen eine Laufzeitbibliothek für eingebettete Systeme. Um verschiedene Einsatzgebiete abzudecken, wurde die Laufzeitbibliothek in Varianten eingeteilt, die sich in Funktion und Umfang unterscheiden. Es existieren zwei Hierarchieebenen: Auf der Ebene der Konfigurationen wurden die CDC (Connected Device Configuration) für Systeme mit mindestens 2 MByte Hauptspeicher, also Geräte im Bereich von Set-Top-Boxen oder Home-Servern, und die CLDC (Connected Limited Device Configuration) für Systeme mit 512 KByte Hauptspeicher, also z. B. Mobiltelefone oder Persönliche Assistenten (PDAs), eingeführt. Die CLDC definiert weiterhin Einschränkungen der virtuellen Maschine selbst, so kann u. a. auf Fließkommazahlen, Finalisation, ein Native-Interface, Klassenlader und Reflection verzichtet werden. Jeder Konfiguration können nun ein oder mehrere Profile zugeordnet werden. Für die CLDC wäre dies z. B. das Mobile Information Device Profile (MIDP), welches es erlaubt, sogenannte MIDlets in einer Sandbox auf einem Gerät zusammen mit anderen Funktionen auszuführen. Dies ist das am weitesten verbreitete Profil, wird es doch auf Mobiltelefonen eingesetzt. Einige Hersteller haben zugunsten von Sonderfunktionen eigene Profile ergänzt, z. B. das Siemens Game API.

3.1.1.1 KVM

Die „Kilobyte“ Virtual Machine (KVM)[6] ist die Referenzimplementierung von *Sun*, sie ging aus einem Projekt für den Palm-Organizer hervor, dem Spotless-System. Die KVM eignet sich z. B. zur Ausführung des MIDP der J2ME. Zusammen mit dem Wireless Toolkit, einer Entwicklungsumgebung, welche unter ähnlichen Konditionen wie das Java Development Kit (JDK) für die J2SE oder J2EE vertrieben wird, ist es möglich, MIDlets zu entwickeln und auf einem Entwicklungssystem in einem Bildschirmfenster zu testen. Die KVM kann von Geräteherstellern als **C**-Quelltext heruntergeladen, an das Zielsystem angepasst und lizensiert werden. *Sun* plant in zukünftigen Versionen die in der J2SE

[6] Die Notation in Anführungszeichen wird von *Sun Microsystems* ebenfalls verwendet, in der Regel wird aber versucht, das Wort „Kilobyte“ zu vermeiden.

und der J2EE verwendete Hotspot-Technologie einzusetzen, was eine höhere Leistungsfähigkeit verspricht aber auch einen höheren Speicherverbrauch nach sich zieht.

3.1.1.2 JBed

Die Schweizer Firma esmertec verfügt mit JBed über eine gegenüber der KVM verbesserten Technologie [8]. Dabei wird dasselbe Zielgebiet (J2ME) abgedeckt, mit einer Ausrichtung sowohl zur CLDC, als auch zur CDC und integrierten Client-Server-Systemen. Neben der reinen Interpretierung von Bytecodes werden verschiedene Compiler-Techniken eingesetzt: Beispielsweise ist es möglich, den Bytecode auf einen Entwicklungssystem in zielsystemspezifischen nativen Code zu übersetzen oder auf dem Zielsystem zur Ladezeit der Klassen. Ein weiterer Compiler verwendet ähnliche Technologien wie *Sun*s Hotspot-Compiler. JBed benötigt zur Ausführung ein vorhandenes eingebettetes Betriebssystem, von denen viele unterstützt werden. Es werden einige Zielsystem-Architekturen direkt unterstützt, u. a. ARM, XScale und C166.

3.1.2 Jamaica VM

Das Karlsruher Unternehmen aicas zielt mit der JamaicaVM auf den Bereich industrieller Anwendungen unter harten Echtzeitbedingungen ab [9]. Die JamaicaVM unterstützt dabei die J2SE 1.4 und die Realtime Specification for Java (RTSJ). Anwendungen für die JamaicaVM werden dabei von einem Entwicklungswerkzeug in **C**-Quelltext konvertiert, übersetzt und zusammen mit der VM zu einer ausführbaren nativen Datei verlinkt. Dadurch werden alle Zielsystem-Architekturen und -Betriebssysteme unterstützt, für die **C**-Compiler existieren. Die harte Echtzeitunterstützung wird durch eine entsprechend ausgelegte Speicherbereinigung ermöglicht [Sie02].

3.1.3 Java-Prozessoren

Neben reinen Software-Projekten existieren auch Ansätze, Java-Bytecode direkt in Hardware auszuführen oder dies zumindest in Hardware zu unterstützen. Prominentestes Projekt in dieser Richtung ist die von *Sun* entwickelte picoJava-Architektur, die in der zweiten Version vorliegt [10]. Es hatten sich jedoch von Anfang an die Schwierigkeiten gezeigt, die bei einer vollständigen Ausführung von Java in Hardware auftreten. Die virtuelle Maschine eignet sich wegen ihrer

komplexen Struktur nicht gut für reale Prozessorarchitekturen. Bei Java treten beispielsweise auf der Ebene der ausführenden Einheit nicht nur einfache Datentypen auf, sondern auch zusammengesetzte Objekte. Auch die Stack-Architektur der ausführenden Einheit verlangt nach neuen Konzepten des Prozessordesigns, werden doch üblicherweise registerbasierte Maschinen verwendet. Die picoJava-Architektur konnte nicht in reale Hardware umgesetzt werden. Andere Ansätze streben daher keine vollständige Java-Maschine in Hardware an, sondern verwenden Kombinationen aus Hard- und Software.

3.1.3.1 aJile

Die von aJile Systems entwickelte Architektur ist in der Lage, Anwendungen der CLDC in Hardware auszuführen [11]. Vertrieben werden sowohl IP-Cores als auch Java-Mikrocontroller mit einigen integrierten Peripheriekomponenten. Anwendungen müssen vorher auf einem Entwicklungssystem vorverarbeitet (optimiert und verlinkt) werden. Bei der Ausführung werden die Java-Stacks auf ein Register-File abgebildet und die Bytecodes mittels Mikrocode abgearbeitet. Direkt unterstützt wird die Isolation mehrerer auf einem System vorhandener Anwendungen; die übrigen Komponenten (Klassenlader, Scheduler, Speicherbereinigung) werden durch eine Laufzeitbibliothek zur Verfügung gestellt.

3.1.3.2 JOP

Die quelltextoffene schlanke Realisierung JOP (Java Optimized Processor) [12] ist in VHDL vorhanden und kann auf kleinen FPGAs (Xilinx Spartan3, Altera Cyclone) synthetisiert werden. Die ausführende Einheit ist eine echte Stack-Maschine, wobei eine Übersetzung der Bytecodes in Mikroinstruktionen oder -instruktionssequenzen erfolgt; komplexe Bytecodes können in Java realisiert werden. Der Stack ist als Cache ausgelegt, welcher nur beim Methodenaufruf mit externem Speicher synchronisiert wird. Die Architektur ist für Echtzeitsysteme ausgelegt und erlaubt prinzipiell eine genaue Vorhersage der Ausführungszeiten (inklusive Stack-Cache). Diese Plattform eignet sich gut für weitere Forschungsprojekte. In [GAKS05] wird ein System für den HW-/SW-Coentwurf vorgestellt, das den JOP als ausführende Einheit benutzt und um anwendungsspezifische Beschleunigungsmodule auf dem FPGA ergänzt. Neben dem System-Bus wird dabei ein Hardware Accelerator Local Bus (HWALB) verwendet, der direkt vom VM-Kern mit speziellen Mikroinstruktionen angesteuert wird. In [GS05] wird eine in Hardware realisierte nebenläufige Speicherbereinigung (Garbage Collection Unit; GCU) für den JOP gezeigt.

3.2 Virtuelle Maschinen für eingebettete 8-Bit-Systeme

Neben den mittlerweile weit verbreiteten und vielfältig vorhandenen virtuellen Maschinen für 32-Bit-Systeme gibt es auch einige, die für 8-Bit-Zielsysteme ausgelegt sind.

3.2.1 Java Card

Für den Einsatz auf Smart-Cards hat *Sun* die Java Card spezifiziert [13]. Dabei kommt eine stark eingeschränkte virtuelle Maschine zum Einsatz, die u. a. keine größeren Datentypen (sogar `int` ist optional), kein Multithreading und keine Speicherbereinigung unterstützt. Zur Unterstützung der Arithmetik mit kleinen Datatypen wurde ein spezieller Bytecode-Satz eingeführt. Ein spezielles Datenformat (JAP) für den plattformübergreifenden Austausch von Anwendungen für die Java Card wurde ebenfalls festgelegt, dabei werden die Klassen teilweise vorverlinkt und Symbole durch interne Tokens ersetzt. Entscheidender Punkt bei der Java Card ist, dass sie nur im Zusammenhang mit einem Card Acceptance Device (CAD), einem Terminal, funktioniert. Wird eine Karte eingeführt, so wählt das CAD die auszuführende Anwendung (Applet) aus. Der Speicher der Java Card ist allerdings persistent ausgelegt, so dass der Zustand der virtuellen Maschine und insbesondere der angelegten Objekte auch im nicht aktiven Zustand erhalten bleibt. Aufgabengebiet für die Java Card ist Kryptographie und die Aufbewahrung und Verwaltung von Datensätzen, wofür eine eigene Programmierschnittstelle existiert. Eine aktive Steuerung externer Komponenten ist nicht vorgesehen.

3.2.2 TINI

Das Tiny InterNet Interface (TINI) von Maxim/Dallas Semiconductors basiert auf dem DS80C400-Mikrocontroller, einem erweiterten und beschleunigten 80C32 [14]. Hierbei handelt es sich allerdings wieder um ein größeres System: Der Adressraum des Controllers wurde auf 24 Bit erweitert und der Registersatz um 32-Bit-Typen ergänzt, so dass größere externe Speicherkomponenten unterstützt werden. Komplettiert wird das System um einen Ethernet-Controller, Einsatzgebiet ist die Anbindung von eingebetteten Systemen an das Internet. Als Betriebssysten dient das eigene TINI OS, welches eine virtuelle Java Maschine mit einer gegenüber der J2SE eingeschränkten und um eigene Komponenten ergänzten Programmierschnittstelle enthält. Dieses System kommt, abgesehen von seiner Größe, dem hier vorgesehenen Einsatzzweck schon recht nahe.

3.2.3 SimpleRTJ

Die von der Australischen Firma RTJ Computing Pty. Ltd. entwickelte virtuelle Maschine SimpleRTJ [15] scheint für den hier vorgesehenen Einsatzzweck gut geeignet zu sein. Die in **C** realisierte interpretierende JavaVM zeichnet sich insbesondere für deren Einfachheit und geringen Speicherverbrauch aus. Dabei verfügt sie über eine nicht nebenläufige Speicherbereinigung und einen einfachen Round-Robin Thread-Scheduler ohne Prioritäten, welcher auf externe Ereignisse (Interrupts) reagieren kann. Java-Anwendungen werden auf einem Entwicklungssystem optimiert und vorverlinkt, das Einbinden von nativen Methoden ist mittels einer eigenen Schnittstelle ebenfalls möglich. Die VM ist skalierbar von 8-Bit-Systemen (68HC11, 8051) bis zu 32-Bit-Systemen (68K, ARM, i386).

3.2.4 LeJOS

LEGO Mindstorms ist ein Baukastensystem, mit dem verschiedene Roboter und Automaten gebaut und in Betrieb gesetzt werden können. Für die zentrale Steuereinheit (RCX), basierend auf dem H8 Mikrocontroller von Hitachi, existieren mittlerweile einige spezialisierte Ersatzbetriebssysteme auf Java-Basis. Die TinyVM [16] ist dabei die einfachste Variante ohne Speicherbereinigung, einem Round-Robin-Scheduler und einigen weiteren Spracheinschränkungen (z. B. `switch`, `instanceof`, `float`). LeJOS [17] ist eine Erweitertrung der TinyVM, welche im Wesentlichen aus einem umfangreicheren API und der Unterstützung von Fließkommaberechnungen besteht. Für LEGO Mindstorms existiert auch noch eine alternative Steuereinheit von Systronics, der JCX [18], welcher auf dem in Abschnitt 3.1.3.1 vorgestellen aJile-VM-Kern basiert.

3.2.5 Muvium

Ein weiteres interessantes Projekt ist die **M**icro[μ]**V**ir**t**ua**l**-**M**achine (muvium) [19], welche auf einem PIC-Mikrocontroller ausgeführt wird. Dabei wird der Java-Bytecode auf dem Entwicklungssystem in nativen Code übersetzt. Es existiert auch ein Interface zur Einbindung eigenen nativen Codes. Die VM verfügt über eine Speicherbereinigung, aber u. a. über kein Multithreading, Datentypen sind auf 16 Bit beschränkt. Viel Wert wird in dem Projekt auf eine integrierte Entwicklungsumgebung (eingenständig oder als Eclipse-Plugin) gelegt, die Teile der auf dem Entwicklungssystem vorhandenen J2SE zusammen mit einer eigenen Bibliothek zur Anwendungserstellung für das Zielsystem verwendet und auch eine Simulation und Remote-Debugging der Anwendungen

erlaubt. Neben Fertiggeräten wird die VM als Software-IP-Core angeboten, um sie auf vorhandene PICs zu laden.

3.2.6 JEPES

Ein compilerbasiertes Projekt, das sich für besonders kleine Zielsysteme eignet, ist JEPES [SBCK03]. Zielsysteme sind der ARM, der Hitachi H8 und x86. JEPES ist eine sehr hardwarenahe Realisierung und unterstützt keine Speicherbereinigung und kein Multithreading, wie bei Java üblich. Stattdessen findet eine Stack-Allozierung der Objekte statt und es werden Interrupt-Routinen unterstützt. Mit Java-Interfaces ist es möglich Konfigurationsdirektiven zu definieren und auf diese Weise einfach und effektiv nativen Code, Interrupt-Vektoren oder die Stack-Allozierung bestimmten Klassen zuzuordnen.

3.3 Diese Implementierung: Yogi2

Einige der vorliegenden VM-Implementierungen kommen schon recht nahe an den hier vorgesehenen Einsatzzweck eingebetteter Steuerungen heran. Neben den zu großen und kostspieligen 32-Bit-Systemen auf J2ME-Basis haben sich einige interessante kleinere VMs gezeigt, die eine passende Funktionalität bieten (SimpleRTJ, muvium). Wesentliche Einschränkung dieser Systeme ist die mangelnde oder fehlende Unterstützung mehrfädiger Anwendungen, was die Integration von Gerätesteuerung, Benutzerinteraktion und Kommunikation auf dem Zielsystem erschwert. Die hier in den weiteren Kapiteln beschriebene Realisierung der Yogi2-VM deckt diesen Punkt bei einer ähnlichen Systemgröße ab.

Die Yogi2-VM ist das Folgeprojekt von Yogi [Böh98]. Dabei flossen die dort gemachten Erfahrungen in die von Grund auf neu entwickelte virtuelle Maschine ein. Probleme, die durch die Wahl von ungünstigen Datenstrukturen und Verarbeitungsschritten bei der Analyse der Java-Klassendateien auftraten, konnten hier vermieden werden. Eine bessere Organisation des Quellcodes ermöglicht nun ein wartbares und flexibles System, das die Integration neuer Konzepte und Verfahren ermöglicht. Der Name von Yogi und Yogi2 geht – wie viele Java-Projekte – auf ein Heißgetränk zurück, den Yogi-Tee® [20].

Kapitel 4

Konzepte für kleine eingebettete virtuelle Java-Maschinen

Wie bereits in Kapitel 3 angedeutet, soll hier die Lücke zwischen dem System geringster Komplexität, der Java Card, und der mittlerweile verbreiteten J2ME geschlossen werden. Primäres Ziel ist, ein System zu schaffen, das mit kaum größerem Aufwand, als dem der Java Card, fast die Funktionalität der J2ME erreicht, wenn auch nicht unbedingt deren Leistungsfähigkeit.

Die zu Beginn des Projekts im Jahre 1998 unter dem Namen Yogi geschaffene Spezifikation für eine 8-Bit-JavaVM [Böh98, BTG98] gilt auch heute noch ohne nennenswerte Änderungen. Lediglich die mit einer 32-Bit-Variante[7] eingeführte Ausdehnung der Systemgröße erfordert nicht mehr die Einschränkungen des Befehlssatzes der virtuellen Maschine. Die im Folgenden vorgestellte Spezifikation der Yogi2-VM bleibt allerdings für Kleinstsysteme optimiert. Eine Skalierung der JavaVM nach unten ist wegen des schlanken Kerns immer leicht möglich. Eine übermäßige Vergrößerung des Systems ist nicht sinnvoll, da hier andere Optimierungen nötig und sinnvoll wären.

Dieses Kapitel stellt die Kernaspekte und -komponenten der JavaVM für kleine eingebettete Systeme zusammen. Sie beruht nur auf der Spezifikation von *Sun Microsystems* für die virtuelle Java-Maschine [LY00] und den Anforderungen an die Zielplattformen, es handelt sich um eine Clean-Room-Implementierung.

4.1 Die virtuelle Maschine als Betriebssystem

Moderne Anwendungen erwarten von den Plattformen, auf denen sie ablaufen, eine gewisse Grundfunktionalität und eine Software-Schnittstelle, um auf die vorhandenen Hardware-Komponenten des Systems zugreifen zu können. Dies leistet gewöhnlich ein Betriebssystem, das einem bestimmten Standard folgt.

[7] Sie soll an dieser Stelle nicht näher betrachtet werden, siehe auch Abschnitt 9.2.

Neben den bei Arbeitsplatzsystemen verbreiteten und bekannten Betriebssystemen hat sich bei eingebetteten Systemen eine große Palette von *Echtzeitbetriebssystem*en (Real Time Operating Systems; RTOS) etabliert.

Der Begriff Echtzeit wird hierbei häufig inflationär verwendet und auch auf Betriebssysteme für eingebettete Systeme angewendet, die keine oder nur sehr bedingte Echtzeitfähigkeiten haben. Die Bandbreite für eingebettete Betriebssysteme ist groß, sie reicht von einfachen Task-Schedulern (z. B. embOS) bis hin zu komplexen Systemen mit Mensch-Maschine-Schnittstelle (z. B. QNX oder VxWorks), die kaum von bekannten Systemen aus der Arbeitsplatzrechnerwelt zu unterscheiden sind; sie entstanden häufig auch aus solchen Systemen oder setzen auf ihnen auf (z. B. RTLinux). Viele Systeme sind in ihrem Funktionsumfang skalierbar und der Komplexität des Zielsystems anpassbar. Folgende Komponenten können von einem Betriebssystem (nicht nur für eingebettete Systeme) verlangt werden:

Prozess-Scheduler (Real-Time-Kernel) Verteilung der Rechenzeit auf die Aufgaben (Tasks). Dabei können verschiedene Algorithmen zum Tragen kommen, die über die Echtzeitfähigkeiten des Systems entscheiden.

Speicherverwaltung Reglementierung des Zugriffs von Prozessen auf den Speicher. Dabei wird es den Aufgaben ermöglicht, den Arbeitsspeicher sinnvoll zu nutzen.

Interprozesskommunikation Gezielter Datenaustausch zwischen den Aufgaben. Der Scheduler kann dadurch beeinflusst werden (Aufgaben warten auf Daten).

Geräteverwaltung Einkapselung systemabhängiger Komponenten in systemunabhängige Aufrufe, Ressourcenzuteilung, Energiemanagement und Steuerung angeschlossener Komponenten (dies ist die eigentliche Aufgabe eines eingebetteten Systems).

Dateisystem Zugriff auf standardisierte Datenträger mit definierten Schnittstellen.

Netzwerkzugriff Kommunikation über Systemgrenzen hinweg (der System- und Anwendungsschicht) mit definierten Protokollen.

Benutzerinteraktion Bedienerführung z. B. mittels einer grafischen Oberfläche.

Diese Komponenten sind zum großen Teil auch in der Sprache Java enthalten. Insbesondere Prozess-Scheduling, Speicherverwaltung und Interprozesskommunikation sind bereits Teil des Sprach-Standards und tief in jede JavaVM eingewoben. Auch die kernelferneren Komponenten existieren bei üblichen Java-

Systemen in Form einer Programmierschnittstelle (API). Hardwarenahe Zugriffe sind damit allerdings nicht oder nur schwierig möglich. Bei Bereitstellung einer geeigneten Programmbibliothek ist Java durchaus in der Lage, die Aufgaben eines Betriebssystems zu übernehmen, ohne seinerseits auf eins angewiesen zu sein. Tabelle 4.1 stellt einige Begriffe aus der Betriebssystem-Welt denen der Java-Welt gegenüber.

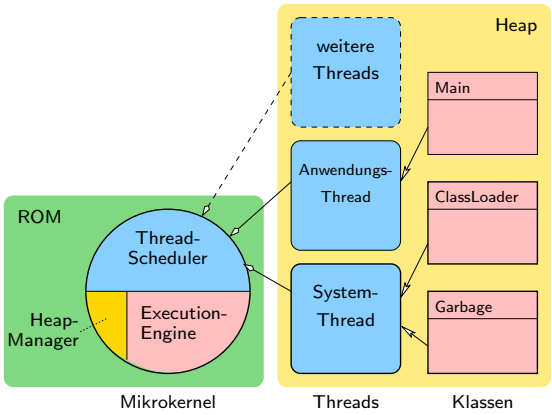
Tabelle 4.1: Gegenüberstellung von Begriffen

Betriebssystem-Begriffe	Java-Begriffe
Prozess, Thread	Thread
`malloc`, `mfree`,	`new`, Speicherbereinigung
Lock, Mutex, Semaphor	`synchronized`
Message	Objekt, EventListener
Datei	Ressource

Die Vorgehensweise, Java auf ein Betriebssystem aufzusetzen, ist bei Systemen, die neben Java auch noch andere Anwendungen ausführen, durchaus sinnvoll. Dies ist z. B. bei den Arbeitsplatzsystemen der Fall, dort ist Java ein eigener Prozess und benutzt die Funktionen des Betriebssystems. Bei eingebetteten Systemen, die nur Java-Anwendungen ausführen, benötigt die zusätzliche Betriebssystemschicht einen vermeidbaren Overhead von Rechenzeit und Speicherverbrauch. Vorteilhaft ist ein Betriebssystem allenfalls beim Portierungsaspekt: Existiert es bereits auf mehreren Plattformen, so ist die JavaVM dort ggf. sogar quelltextkompatibel.

Java bietet gegenüber üblichen Betriebssystemen noch weitere Möglichkeiten, Java ist objektorientiert und stellt somit ein *objektorientiertes Betriebssystem* (OOOS) zur Verfügung.

Die Struktur der Yogi2-VM ist in Bild 4.1 skizziert. Wie auch in vielen Echtzeit-Betriebssystemen, wird ein Mikrokernel-Konzept verwendet. Ziel dabei ist, den Teil der JavaVM, der immer auf einem Zielgerät präsent sein muss, möglichst klein zu halten. Alle anderen Komponenten, die nicht zum Ausführen von Bytecodes, zur Speicherverwaltung oder zur Zuteilung von Rechenzeit nötig sind, sind optional und können bei Bedarf hinzugefügt werden. Solche Komponenten sind z. B. Klasseninitialisierung und Speicherbereinigung, die von einem gewöhnlichen Java-Thread ausgeführt werden und vom Kern auch genauso wie ein Anwendungs-Thread behandelt werden.



Erstmalig publiziert in [BT01a].

Bild 4.1: Die Struktur der Yogi2-VM

4.2 Konzepte für eine speichereffiziente VM

Wie gezeigt, ist Java in der Lage, die Grundfunktionalität eines Betriebssystems zu übernehmen. Bleibt die Frage offen, ob Java nicht kontraproduktiv bzgl. des Ressourcenverbrauchs ist, sind doch bereits kleine Java-Anwendungen im Arbeitsplatz-Bereich als speicherplatzhungrig bekannt. Dies hat zwei Gründe:

1. Hinter einfachen Aufrufen des Java-APIs verbergen sich häufig größere Vorgänge, die insbesondere bei Anwendungen mit grafischen Bedienoberflächen viele Laufzeitobjekte erzeugen. Die Komplexität von Anwendungen steigt, ohne dass dies vom Programmierer bemerkt wird. Durch geeignete Programmiertechniken und ein angepasstes API ist es möglich, den Laufzeitspeicherverbrauch zu verringern (siehe Abschnitt 8.2 und [BT02]).

2. Eine Java-Anwendung liegt nicht als in sich geschlossenes Executable (Programm-Datei) vor, sondern in Form der einzelnen Klassendateien (siehe Abschnitt 2.1). Die zum Verlinken nötigen Zusatzinformationen (Symbole) sind in den Klassendateien zusätzlich zum eigentlichen Bytecode enthalten. Wird der Link-Zeitpunkt vorverlagert und von dem eingebetteten System ferngehalten, so können dort diese Symbole entfallen und der Festwertspeicherverbrauch für Programme verringert sich.

Bei Arbeitsplatzsystemen finden Programme und Laufzeitdaten im großzügig bemessenem Laufzeitspeicher Platz, die Klassendateien werden vorher vom dort langsameren Dateisystem kopiert und ggf. entkomprimiert. Kleine eingebettete Systeme unterscheiden gewöhlich zwischen Festwertspeicher für Programmcode und Laufzeitspeicher für Daten. Da der Laufzeitspeicher sehr klein ist (gewöhnlich ist er eine Größenordnung kleiner, als der Festwertspeicher), würde er auch die Größe von Java-Anwendungen begrenzen. Stattdessen werden die Klassen gemäß des in Abschnitt 2.2 vorgestellen Vorgehens in einem direkt ausführbaren Abbild im Festwertspeicher abgelegt, unkomprimiert aber gekürzt und optimiert. Dies soll in Abschnitt 4.2.5 noch genau betrachtet werden.

Bevor im Folgenden die Mechanismen im Kern der virtuellen Maschine dargestellt werden, um Java-Klassen speichereffizient abzulegen und auszuführen, sollen einige Speicherplatz-Effekte, die den Java-Bytecode selbst betreffen, beleuchtet werden.

4.2.1 Speicherspareffekte von Java

Bei Java-Bytecode handelt es sich um eine sehr kompakte Repräsentation von Algorithmen. Interessanterweise entstand der Bytecode zu einer Zeit, als sich der bis heute fortgesetzte Trend zu breiteren Prozessorarchitekturen mit 32- oder 64-Bit-aligned RISC-Instruction-Sets deutlich abzeichnete. Diese haben auf Systemen mit schnell angebundenem Speicher ihre Vorteile, da die Dekodierung der Opcodes und der Speicherzugiff selbst vereinfacht wird. Somit kann Bytecode mit seinen variablen Parametergrößen, selbst wenn man von dem Interpreter-Overhead absieht, auf diesen modernen Architekturen nicht ohne eine Übersetzung in nativen Code effizient ausgeführt werden. Anders sieht das bei kleinen 8-Bit-Systemen aus. Untersuchungen haben gezeigt, dass – je nach Anwendungsfall – die meisten verwendeten Opcode-Parameter (Immediate, Displacement, Branch) mit einem Byte auskommen [HP94, Kap. 3.4ff]. Java-Bytecode kommt diesem Optimum schon recht nahe (siehe Tabelle 4.2). Einen großen Beitrag dazu leistet die Architektur der JavaVM als Stackmaschine, so dass z. B. die Übergabe von Registern als Parameter entfällt (gelegentlich müssen dann natürlich auch Stack-Manipulationen vorgenommen werden, die einen Teil dieser Einsparungen wieder aufbrauchen).

Noch extremer fällt dieser Vergleich aus, wenn als Zielplattform ein kleines 8-Bit-System betrachtet wird. Hierbei sind die von Java beschriebenen Algorithmen nur mit sehr viel mehr Anweisungen umsetzbar als Bytecodes erforderlich sind. Dies liegt an der Wortbreite der Register, denn um mit diesen Systemen eine 16- oder 32-Bit-Operation durchzuführen, müssen mehrere 8-Bit-Anweisungen verkettet werden. Der kompakte Bytecode bietet gegenüber dem

Tabelle 4.2: Java-Bytecode-Klassen

Bytecode-Klasse	Parameter-Bytes	relative Häufigkeit [%]*
Konstante laden	0	9,7
	1	12,0
	2	1,3
lokale Variable laden/speichern	0	17,0
	1	9,5
Objekt-Variable laden/speichern	2	11,4
Stack-Manipulation	0	4,5
Arithmetik	0	6,6
	2	1,1
(bedingter) Sprung	2	7,7

*Bezogen auf alle Bytecodes im Test-Satz, ermittelt durch die Entwicklungswerkzeuge anhand des Beispielprogramms `SystemSetup`, das 7924 Bytecodes umfasst; die genaue Auflistung befindet sich in Anhang E.1.

nativen Befehlssatz des verwendeten Mikrocontrollers eine Kompression – je nach Anwendungsfall – um einen Faktor bis zu sechs. Dieser Faktor ergibt sich beispielsweise bei der Umsetzung des Algorithmus

```
y=(short)(i*x+i+i-i/x>>2),
```

welcher in 15 Byte Java-Bytecode umgesetzt wird, aber eine Assembler-Sequenz von 90 Byte erfordert (bei der Verwendung von 16-Bit-Worten, siehe unten; mangels Divisionsbefehl wird für die Division eine Unterroutine aufgerufen, die in diese Rechnung nicht mit eingeht). Der komplette Vergleich hierzu ist in Anhang E.2 zu finden. Die Bytecode-Interpretierung bietet also eine Dekompression von Algorithmen zur Laufzeit, ohne eine Zwischenspeicherung im Laufzeitspeicher zu benötigen. Wie effektiv diese Komprimierung ist, wurde in [CSCM00] gezeigt. Dort wurde mit Faktorisierungstechniken eine weitere Komprimierung des Bytecodes vorgenommen, wobei er mit einer angepassten und makrofähigen JavaVM nur mit geringem Performanceverlust ausführbar bleibt. Es zeigte sich eine weitere Platzersparnis von lediglich 15% (mit dem Gzip-Algorithmus wurde ein vergleichbarer Wert erreicht). Die Verwendung von Java ermöglicht es, auf kleinen Systemen umfangreichere und komplexere Anwendungen unterzubringen, als es mit nativem Code bei gleicher Speicherausstattung möglich ist, hinzu kommen noch die Vorteile der Objektorientierung von Java, die die Erstellung komplexer Anwendungen erleichtert.

4.2.2 Eingeschränkter Befehlssatz und Datentypen

Zusätzlich zu den Java-Anwendungen muss die virtuelle Java-Maschine auf dem Zielsystem Platz finden. Damit dies kompakt möglich ist, können Einschränkungen gegenüber der VM-Spezifikation vorgenommen werden, die den Betrieb eingebetteter Anwendungen nicht wesentlich behindern.

Die virtuelle Java-Maschine ist für einen Satz von acht primitiven Datentypen spezifiziert, dabei kommen auch Fließkomma- und 64-Bit-Datentypen vor, die auf einem 8-Bit-System nur mit großem Aufwand bearbeitet werden können. Dieser Aufwand besteht zum einen aus Programmtext, da der Befehlssatz dieser Systeme solche Datentypen in der Regel nicht unterstützt, und zum anderen aus der benötigten Rechenzeit für die Emulation der Befehle. Ferner werden diese Datentypen im Einsatzfeld von 8-Bit-Systemen (der Erfassung von Messwerten, einfache Steuerungen und Kommunikation) eher selten benötigt, es finden keine umfangreichen Berechnungen statt. Daher liegt ein Verzicht auf diese Datentypen bei der VM-Implementierung nahe. Die zur Unterstützung dieser Datentypen notwendigen Bytecodes für arithmetische Berechnungen und Typwandlung können in der VM-Implementierung ausgelassen werden.

Die genannten Einschränkungen sind in der Lage, die Größe des VM-Kerns, also des erforderlichen Festwertspeichers zu reduzieren. Der knappe Laufzeitspeicher bleibt davon allerdings unberührt. Eine weitere nicht spezifikationsgemäße Einschränkung kann diesen Speicher ebenfalls reduzieren, die Reduktion der *Wortbreite* von 32 auf 16 Bit. Dies ist möglich, da die Spezifikation der virtuellen Java-Maschine keine Vorschriften über die Größe eines Wortes macht, außer dass der Referenztyp, der sich auf Objekte bezieht, darin Platz finden muss. Bei Mikrocontrollern, die über 64 KByte Adressraum verfügen, sind dies lediglich 16 Bit. Problematisch ist in diesem Zusammenhang, dass der Typ `int` zu 32 Bit festgelegt wurde und ebenfalls in einem Wort Platz finden muss. Es kommt also bei der Ausführung von Java-Anwendungen, die diesen Datentyp benutzen, zu Abweichungen vom spezifizierten Verhalten. Normalerweise kann diese Abweichung auch nicht zur Übersetzungs- oder Ladezeit einer Anwendung festgestellt werden, sondern fällt u. U. erst zur Laufzeit auf. Dieses Missverhalten der VM muss der Anwendungsprogrammierer also kennen und berücksichtigen. Diese Einschränkung kann den notwendigen Laufzeitspeicher für eine Anwendung fast halbieren, so dass sie hier dennoch bei der Implementierung von 8-Bit-VMs angewendet wird. Die Berechnungen können dann ebenfalls von 32 auf 16 Bit reduziert werden, was also eine weitere Ersparnis von Code der VM-Implementierung zur Folge hat.

Die Beschränkung auf einen 64 KByte großen Adressraum hat noch einen weiteren Effekt, der zu einer minimalen Code-Einsparung führt: in den Bytecodes

der Anwendungen kann kein *wide-Opcode* vorkommen. Dieser Bytecode hat eine Shift-Funktion für den nachfolgenden Bytecode und wird z. B. für bedingte und unbedingte relative Sprünge verwendet, die nicht in einem 16-Bit-Bereich stattfinden. Da die Größe der Java-Klassen beschränkt ist, kann dieser Bereich nicht verlassen werden. Die Implementierung der alternativen Varianten dieser Bytecodes kann also in der VM-Implementierung ohne Auswirkungen auf die Anwendungen entfallen.

Viele Bytecodes greifen auf den Konstantenpool einer Klasse zu. Eine Implementierung einer 8-Bit-VM kann die Anzahl der Konstantenpool-Einträge beschränken, um den Laufzeitspeicherverbrauch und die Größe von Zugriffstabellen im Festwertspeicher zu reduzieren. Vorgegeben ist ein Bereich, der durch 16 Bit erfasst werden kann und hier auf 8 Bit beschränkt wird. Für Zugriffe auf den 16-Bit-Bereich des Konstantenpools werden ebenfalls spezielle Bytecodes mit einer `wide`-Erweiterung verwendet, die nicht implementiert werden müssen.[8]

4.2.3 Datenstrukturen

Eine JavaVM führt Bytecode in Java-Klassen aus. Als Austauschmedium zwischen Compiler und VM dient das Class-File-Format, welches in der Spezifikation für die virtuelle Java-Maschine genau festgelegt ist. Dieses Format ist für den Transport der Klasse vorgesehen und nicht direkt für die VM geeignet, den enthaltenen Code auszuführen und auf Daten zuzugreifen. Eine Klasse liegt dort als Datenstrom ohne direkte Querverweise vor, sämtliche Referenzen auf die Elemente der eigenen und anderer Klassendateien finden symbolisch (textuell) statt. Das hat den Vorteil, dass ein Software-System in Java bei Änderungen nicht komplett neu übersetzt werden muss, sondern es auch mit einzelnen Teilen funktioniert. Der Nachteil dabei ist, dass der virtuellen Maschine auferlegt wird, diese Daten geeignet zur Ausführung aufzubereiten (*Late Binding*). Im Folgenden werden die Datenstrukturen der Yogi2-VM vorgestellt und die Beziehungen zur entsprechenden Klassendatei dargestellt.

[8] Hierbei gibt es Ausnahmen: Bei Verkürzung des Konstantenpools vom Vorverlinker (siehe Abschnitt 4.2.5) können `wide`-Erweiterungen im Bytecode zurückbleiben, obwohl der Konstantenpool den 8-Bit-Bereich nicht überschreitet. Weiterhin kann ein Java-Compiler entscheiden, aus Optimierungsgründen auch dann einen `wide`-Bytecode zu verwenden, wenn der erweiterte Tabellenzugriff nicht nötig ist. So ermöglicht z. B. die `wide iinc`-Instruktion eine 16-Bit-Konstantenaddition auf eine lokale Variable, was normalerweise nur über den Java-Stapel und mit mehreren Bytecodes möglich ist.

4.2.3.1 Die Repräsentation des Konstantenpools

Der Konstantenpool ist die wichtigste und größte Datenstruktur der Klassendatei. Er enthält nicht nur die im Programm spezifizierten Konstanten und String-Literale, sondern auch sämtliche symbolischen Referenzen, die sich aus der Struktur des Programms ergeben, z. B. Variablen- und Methodennamen. Der Konstantenpool liegt als ein Datenstrom mit Einträgen unterschiedlicher Größe vor, so dass ein einzelner Eintrag nur erreicht werden kann, wenn zuvor alle vorherigen Einträge analysiert wurden. Zur Laufzeit der VM ist also eine Tabelle für einen direkten Zugriff angebracht, die *Constant Pool Representation* (CPR), dessen reguläre Struktur Bild 4.2 skizziert.

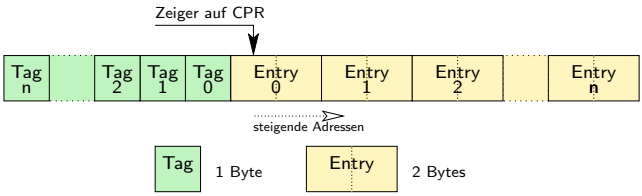


Bild 4.2: Die Constant Pool Representation (CPR)

Diese CPR hat genauso viele Einträge wie der Konstantenpool der Klassendatei und besteht aus zwei Teilen:

1. Die Entries (Einträge) enthalten die eigentliche Information aus dem Konstantenpool, folgende Typen werden unterschieden:

   Zeichenketten werden durch einen 16-Bit-Zeiger referenziert, dabei wird auf den originalen Konstanenpool verwiesen (näheres dazu in Abschnitt 4.2.5).

   Klassen und Strings verweisen ebenfalls direkt auf die Zeichenkette.

   Konstanten stehen direkt im Eintrag.

   Verweise auf Variablen und Methoden enthalten Tabellenindizes auf den Variablenbereich eines Objekts oder einer Klasse bzw. in die Methodenreferenztabelle einer Klasse (sie wird im nächsten Abschnitt betrachtet).

2. Die Tags (Markierungen) sind jeweils genau einem Entry (Eintrag) zugeordnet und stammen ursprünglich aus dem Konstantenpool, um den Typ und die Größe des Eintrags zu bestimmen. Die Größeninformation wird

hier nicht mehr benötigt und die Typ-Information des Tags ist redundant, stattdessen wird hier Zusatzinformation zum Entry abgelegt:

positive Werte 0...127 (ursprünglicher Wert aus der Klassendatei): der Eintrag enthält irrelevante oder unbekannte Daten, dies sollte im Betrieb der VM nicht vorkommen bzw. eine Sonderbehandlung erfahren (der Konstantenpool-Eintrag wird zur Laufzeit der VM aufgelöst).

negativer Wert -128: der Eintrag verweist auf eine Zeichenkette.

negativer Wert -1: der Eintrag enthält gültige Daten.

negative Werte -2...-127: wie bei -1, zusätzlich wird eine Laufzeitklassenreferenz angegeben, dies ist bei Einträgen vom Typ Klasse oder Verweis relevant, siehe Abschnitt 4.2.3.3.

Die CPR wird durch einen Zeiger adressiert, der auf den ersten Entry zeigt. Die Entries erstrecken sich in positiver Richtung (steigende Adressen) und die Tags in negativer Richtung. Das ist eine optionale Optimierung, um Zeiger bzw. Zeiger-Arithmetik einzusparen. Der Zugriff erfolgt in beiden Fällen per indizierter indirekter Adressierung, die auf sehr vielen Prozessorfamilien verfügbar ist.

4.2.3.2 Die Methodenreferenztabelle

Die Objektorientierung der Programmiersprache Java ist bis auf die Ebene der virtuellen Maschine sichtbar. Der VM selbst ist es auferlegt, z. B. bei dem Aufruf einer virtuellen Methode die richtige Klasse und den richtigen Bytecodeeinsprungpunkt zu finden. Die Klassendateien liefern der VM dazu nur die nötigsten symbolischen Informationen, es werden keine geeigneten Datenstrukturen vorgeschrieben. Sinnvoll ist hier eine hierarchische Tabellenstruktur, die für jede Klasse alle erreichbaren Methoden enthält und die vererbten und überschriebenen Methoden auflöst. Bild 4.3 zeigt die *Methodenreferenztabelle* (Method Reference Table, MRT) in einer Klasse, sie ist ähnlich aufgebaut, wie die CPR. Das Flag-Feld enthält die für die Ausführung der Methode notwendigen Informationen:

- die Größe der Parameterliste (Anzahl der lokalen Variablen), diese Information ist normalerweise nur symbolisch im Methodendeskriptor enthalten und liegt hier für einen schnellen Zugriff dekodiert vor,
- ist die Methode `synchronized`, so muss beim Aufruf ein Monitor-Erwerb eingeschoben werden und dieser auf den Stack gesichert werden,

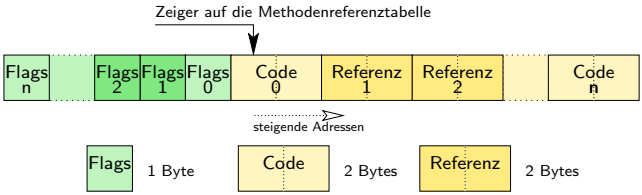


Bild 4.3: Die Methodenreferenztabelle

- ist die Methode vererbt, so muss die entsprechende Superklasse, die den Code enthält, aufgefunden werden. Dieses Flag entscheidet, ob in dem Haupteintrag der Methode ein Zeiger auf den Code in dieser Klasse enthalten ist oder eine Referenz auf den entsprechenden Tabelleneintrag in der Superklasse (dies ist in Bild 4.3 durch unterschiedliche Farben angedeutet).

Das letzte Flag ist ein Kompromiss, um den Speicherplatzbedarf der Tabelle zu reduzieren. Ohne dieses Flag müssten je Eintrag ein Code-Zeiger und eine Klassen-Referenz vorhanden sein. Der Nachteil bei dieser Einsparung ist eine verminderte Zugriffszeit auf abgeleitete Methoden, da der Zugriff in diesem Fall mit einer zusätzlichen Ebene Indirektion erfolgt. Ein Vorteil ist allerdings eine bessere Einkapselung der Klassen. So müssen die Zeiger auf den Code der Methoden anderen Klassen nicht bekannt sein, was vor allem beim späteren Hinzuladen von Anwendungsklassen eine Rolle spielt (siehe auch Abschnitt 4.2.5). Methodenmodifikatoren wie `private`, `protected` oder `final` finden sich an dieser Stelle nicht mehr und werden von der VM nicht ausgewertet. Die Zugriffsbeschränkungen werden stattdessen beim Erstellen dieser Tabellen geprüft, also vor der Ausführungszeit des Codes.

Bild 4.4 zeigt im Beispiel die Methodenreferenztabellen von drei abhängigen Klassen, die dem *JControl*-API entnommen sind. Wegen der Übersichtlichkeit werden die Flags mit den Haupteinträgen zusammen gezeichnet. Folgende Eigenschaften können festgestellt werden:

- Die Reihenfolge der Methoden in einer Superklasse wird in die Subklassen übernommen, diese ergänzen lediglich weitere Methoden. So ist ein kompatibler Zugriff auf die Methoden von Objekten möglich, von denen zur Übersetzungszeit lediglich der Superklassentyp bekannt war (Auflösung der Vererbung).
- Vererbte Methoden referenzieren direkt die Methoden in den Superklassen, in denen sie implementiert wurden, so ist ein Zugriff auf jede Methode

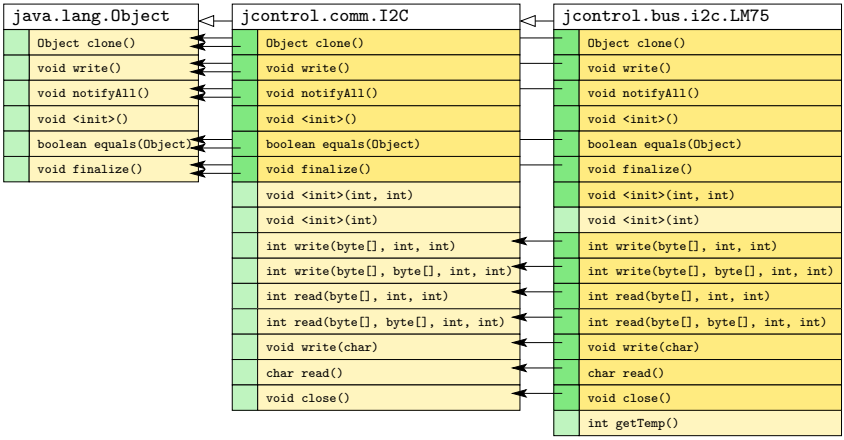


Bild 4.4: Beispiel zur Methodenhierarchie

mit maximal zwei Tabellenzugriffen möglich.

- Subklassen können vererbte Methoden überschreiben, in diesem Fall wird die Referenz in die Superklasse durch einen Zeiger in den eigenen Code ersetzt (Auflösung der Vielgestaltigkeit).
- Es ist keine Unterscheidung zwischen statischen und Instanzmethoden nötig.
- Konstruktoren sind der Einfachheit halber in diese Struktur eingereiht, die JavaVM muss nicht zwischen Konstruktoren und gewöhnlichen Methoden unterscheiden. Da Konstruktoren nicht vererbt werden können und dürfen, entfallen hier die Referenzen in die Superklassen.
- Eine Optimierung ist es, die Methoden, die nicht vererbt werden können (`private` Methoden und Konstruktoren), jeweils ans Ende der Tabelle einer jeden Klasse zu schreiben, diese werden dann vor der Vererbung in eine Subklasse abgeschnitten und belegen nicht unnötig Tabellenplatz.

*Java-Interface*s können über diese Hierarchie nicht aufgelöst werden, sie verfügen über eine komplett eigene Vererbungshierarchie, die prinzipiell genauso strukturiert ist wie die Methodenhierarchie von Klassen. Da in Interfaces kein Code definiert werden kann, müssen diese Tabellen der JavaVM nicht zur Verfügung gestellt werden, nur die jeweils letzte Ebene in der implementierenden Klasse. Die Tabelle oder die Tabellen (falls mehrere Interfaces imple-

mentiert werden), werden dort einfach an die Klassenmethodenreferenztabelle angehängt. Dabei wird die komplette Interface-Hierarchie in die Liste der implementierten Interfaces in die Klasse übernommen, nicht nur die explizit von der Klasse implementierten Interfaces. Dies befreit die VM von der Auflösung der Interfaces komplett, so dass die Interfaces in Form von übersetzten Klassendateien nicht benötigt werden.[9] Die Liste der implementierten Interfaces hat dieselbe Struktur wie ein Methodenverweis in der CPR. Um den Index in der Methodenreferenztabelle der Interface-Methode zu lokalisieren, werden beide Werte zusammengezählt. Die Interfaceeinträge in der Methodenreferenztabelle haben denselben Aufbau, wie die Methodeneinträge. Einige Besonderheiten sind zu beachten:

- Da Interfaces keinen Code enthalten, sind nur Einträge vom Referenz-Typ vorhanden.
- Die Verweise zeigen nicht nur auf eine Superklasse, sie können auch auf die eigene Klasse zeigen.
- Die Liste der implementierten Interfaces in der Klasse verweist für jedes Interface auf den entsprechenden Anfangseintrag in der Methodenreferenztabelle.

Bild 4.5 zeigt diese Zusammenhänge, zu beachten ist hier, dass die Interface-Hierarchie selbst nicht gespeichert wird. Im Gegensatz zur Hierarchie gewöhnlicher Klassen hat sie nicht nur eine Wurzel (`java.lang.Object`).

4.2.3.3 Laufzeitreferenzen auf dem Java-Heap

Alle bisher genannten Tabellen, die die JavaVM benötigt, sind statisch, d. h. sie ändern sich während der Laufzeit nicht. Jede gerade benutzte Klasse benötigt noch einen Laufzeitdatensatz, der sie als Java-Objekt verfügbar macht. Er befindet sich wie alle Java-Objekte auf dem Heap, der in Kapitel 5 noch genauer beschrieben wird.

Diese Blöcke des Typs `Class`[10] enthalten zusätzlich die *Laufzeitklassenreferenz*en. Zunächst dient ein Klassenobjekt als Container für klassenspezifische

[9] Für einen Fall werden die Interface-Klassen dann allerdings doch benötigt: Cast-Checks. Dabei wird implizit durch das `checkcast`- oder `instanceof`-Kommando die Interface-Klasse geladen, nur um festzustellen, ob es sich um ein Interface handelt.

[10] Ein Block des Typs `Class` ist nicht mit einer bei Java bekannten Instanz vom Typ `java.lang.Class` zu verwechseln, welche den Zugriff auf die Elemente einer Klasse zur Laufzeit ermöglicht. Damit können z. B. Listen von Variablen und Methoden einer Klasse erstellt werden (Reflection-Mechanismus). Diese Funktionalität ist aus Speicherplatzgründen bei dieser 8-Bit-VM nicht implementiert.

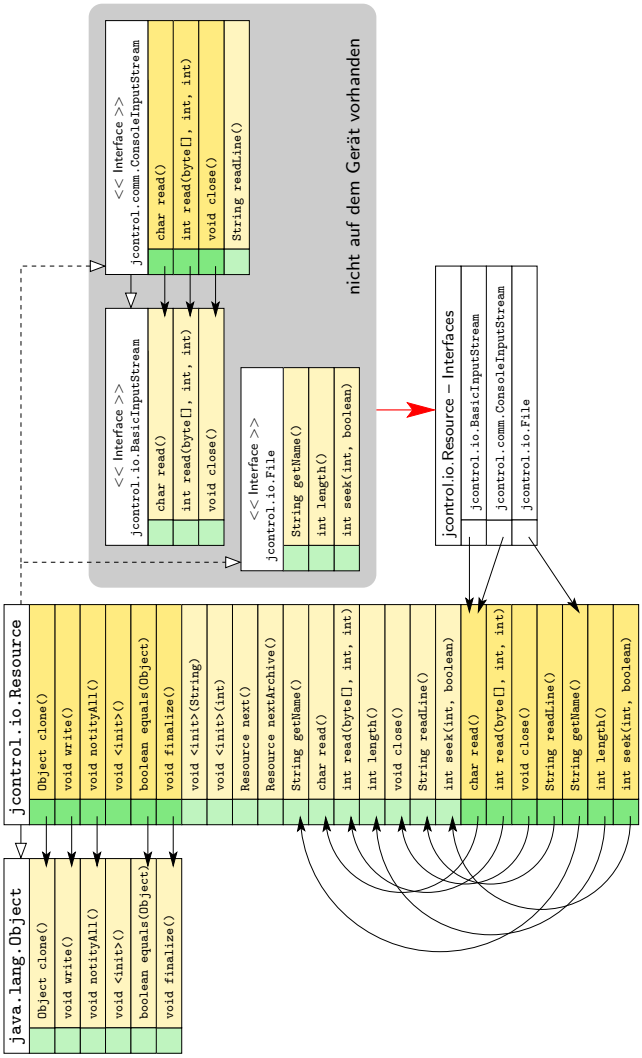


Bild 4.5: Integration von Interfaces

Daten wie Superklasse und Interfaces sowie für die oben in Abschnitt 4.2.3.1 und 4.2.3.2 genannten statischen Strukturen CPR und MRT. Dabei sind die Strukturen nicht direkt im Klassenobjekt enthalten, sondern im Festwertspeicher und es wird nur mit Zeigern auf sie verwiesen. Ist eine Klasse vollständig initialisiert, so ändern sich diese Tabellen nicht mehr zur Laufzeit der VM. Die Struktur der Klasse wird einmalig durch den Compiler vorgegeben. Anders ist das nur bei den Referenzen auf andere Klassen, dabei handelt es sich um Objekt-Referenzen, die zur Laufzeit (also abhängig vom Programmfluss der Java-Anwendung) entstehen. Diese Referenzen werden zu einer Tabelle zusammengefasst. Zugegriffen wird auf diese Referenzen über den Tabellenindex, der sich in der CPR befindet. Die Position der Einträge ist fest. Diese Tabelle ist die einzige Struktur, die die VM zur Laufzeit verwalten muss und die daher wertvollen Heap-Speicherplatz belegt. Alle übrigen Klassenstrukturen können im Festwertspeicher abgelegt werden. Im Klassen-Objekt auf dem Heap existieren lediglich Verweise darauf.

4.2.4 Klassendateien auf dem Zielsystem

Wie schon mehrfach erwähnt, wird Java-Bytecode bei kleinen eingebetteten Systemen sinnvollerweise direkt aus dem Festwertspeicher ausgeführt. Organisiert sind die Klassen dort in eigenen Strukturen, die mittels einer einfach verketteten Liste zu einem Archiv zusammengestellt werden. Eine vergleichbare Einheit bei Java auf Arbeitsplatzsystemen ist das Java-Archiv (JAR). Ziel des eingebetteten Archiv-Formats soll auch hier die Speicherplatzeffizienz sein und nicht eine mögliche Kompatibilität zu anderen Systemen. Der Festwertspeicher ist nicht zur Verwendung von außerhalb vorgesehen. Eine Redundanzkomprimierung des Archivinhalts findet wegen des gewünschten Direktzugriffs auf die Bytecodes nicht statt. Es kann allerdings eine Irrelevanzreduktion der Daten der Klassendateien vorgenommen werden (siehe nächster Abschnitt).

Festwertspeicher bei eingebetteten Systemen kann grundsätzlich in einmal programmierbaren Speicher (One Time Programmable ROM; OTP) und wiederprogrammierbaren Speicher (Flash-EEPROM) eingeteilt werden. Ersterer eignet sich für die virtuelle Java-Maschine und die API-Klassen der Laufzeitbibliothek. Anwendungen können auch im ROM liegen, bei vielen Geräten ist allerdings der Austausch von Anwendungen sinnvoll, was insbesondere in der Entwicklungsphase eine Rolle spielt (Rapid Prototyping). Auch technologische Gegebenheiten können eine Aufteilung in änderbaren und nicht änderbaren Festwertspeicher erfordern, wenn der verwendete Mikrocontroller selbst nicht über wiederprogrammierbaren Speicher verfügt und dieser extern über Ports als

Datenspeicher angeschlossen werden muss. Da dort kein nativer Code ausgeführt werden kann, ist das auch ein Sicherheitsmechanismus: Ein Java-Entwickler kann den Kern des Systems nicht verändern, insbesondere kann kein Code eingebunden werden, der direkt auf den Speicher des Controllers zugreift.

Bei der hier durchgeführten konkreten Implementierung der Yogi2-VM verfügt der Mikrocontroller über 60 KByte internen OTP-Speicher, der den VM-Kern und die Laufzeitbibliothek aufnimmt. Optional kann ein Flash-Speicher angeschlossen werden, der in Bänke zu je 64 KByte aufgeteilt ist. Jede Bank kann inviduell mit Java-Anwendungs-Archiven gefüllt werden. Jeweils eine Bank wird beim Start der VM ausgewählt (aktive Bank) und die darin befindliche Anwendung gestartet. Zur Laufzeit einer Anwendung sind die anderen Bänke des Flash-Speichers ausgeblendet. Es gibt also nur eine interne Bank und eine externe Bank, welche jeweils einen eigenen Adressraum darstellen. Zur Adressierung verfügen die VM-internen Zeiger-Variablen über ein zusätzliches Flag (ein 17. Adressbit). Dieses Flag wird einmalig beim Zugriff auf die Klasse gesetzt und kann, solange nicht der Klassenkontext verlassen wird, so verbleiben, da die Daten einer Klasse komplett innerhalb eines Archivs und somit einer Bank liegen. Bei VM-interner Zeigerarithmetik wird das Flag nicht verändert, es wird lediglich kopiert, falls nötig.

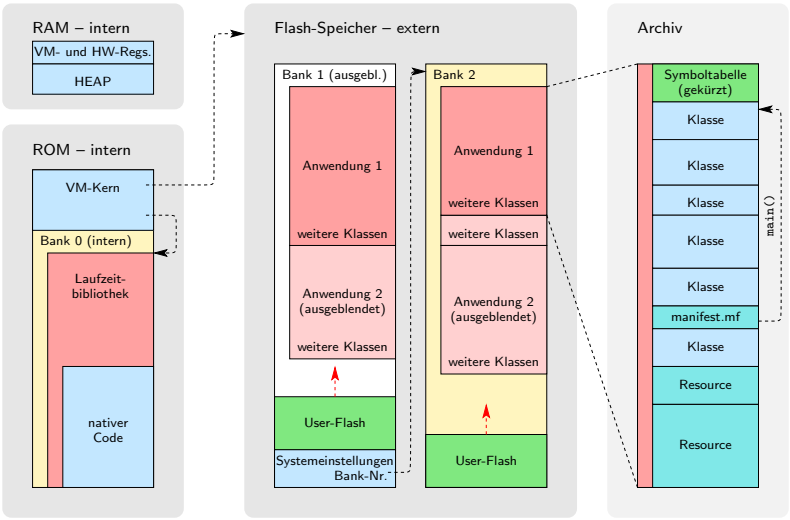


Bild 4.6: Die Verteilung von Archiven auf Bänke

Bild 4.6 zeigt grob eine Verteilung von Archiven auf die Bänke. Im VM-Kern existieren lediglich zwei feste Zeiger auf die interne Bank und den externen Flash-Speicher. Dort befindet sich am Ende von Bank 1 ein Bereich mit Systemeinstellungen, wo die für die Laufzeit der VM aktive externe Bank vermerkt ist. Im Kopf jeder externen Bank befindet sich eine Liste der vorhandenen Archive, die *Segmentliste*, die auch zu jedem Archiv einen Namen und Zugriffsflags speichert. Die Abfolge der Archive in der Segmentliste definiert zusammen mit der internen Bank eine Suchreihenfolge für Klassen und Ressourcen. Auf Arbeitsplatz-Java-Systemen ist eine vergleichbare Information in der Umgebungsvariable *`Classpath`* gespeichert. Die Suchreihenfolge in der Segmentliste muss nicht mit der physikalischen Reihenfolge der Archive übereinstimmen. Zum Ändern der Suchreihenfolge muss nur die Segmentliste neu erstellt werden.

Jedes Archiv ist in sich abgeschlossen und enthält keine direkten Referenzen über seine Grenzen hinaus. Die Klassen im Archiv sind hingegen nicht abgeschlossen. Anders als es sich in der Zeichnung darstellen lässt, ist die symbolische Information aller Klassen über das gesamte Archiv verteilt und zusammengefasst. Dabei findet eine Redundanzminimierung statt, da nun jedes Symbol in einem Archiv nur einmal vorkommt. Die Symboltabelle ist bzgl. der Archive abgeschlossen, nicht mehr bzgl. der einzelnen Klassen. Anstatt Referenzen über gleiche Symbole in den einzelnen Klassen zu ermitteln, findet der Zugriff über direkte bankweise Adressierung statt. Symbole werden nur noch benötigt, um Klassen in anderen Archiven anzusprechen und um Methoden zu suchen, deren Name erst zur Laufzeit ermittelt werden kann. Weiterhin werden in den Archiven die symbolischen Namen der Klassen selbst benutzt, um sie im Archiv zu finden. Lediglich bei Ressourcen, die keine Klassen sind, ist ein zusätzliches Symbol nötig. Ressourcen werden direkt als Binärabbild abgelegt. Weitere Informationen zu den hier verwendeten Datenstrukturen der Yogi2-VM finden sich in Anhang C.3.

Zum Starten einer Java-Anwendung wird dasselbe Verfahren angewendet wie bei JARs. Es wird entsprechend der Suchreihenfolge die Ressource `META-INF/manifest.mf` lokalisiert und dieser Textdatei der Name der zu startenden Klasse entnommen und ggf. noch weitere Startparameter. Die Klasse wird ebenfalls entsprechend der Suchreihenfolge ermittelt (es kann sich also auch um ein anderes Archiv handeln). Sie wird initialisiert und dort dann die Methode `main()` in einem neuen Thread aufgerufen. Auf der Anwendungsebene kann auf die in den Archiven vorhandenen Ressourcen zugegriffen werden. Klassen können hingegen nicht ausgelesen werden, das Erstellen eines Inhaltsverzeichnis eines jeden Archivs ist aber möglich.

VM und Anwendungen haben nur einen lesenden Zugriff auf die Archive. Veränderungen sind nur mittels eines von außen einwirkenden Downloaders möglich.

Der von den Archiven und den Systemeinstellungen nicht genutzte Flash-Speicherplatz wird der Anwendungsschicht zur Verfügung gestellt. Dort besteht die Möglichkeit eines blockweisen Schreib-/Lesezugriffs, wobei die von den Archiven belegten Blöcke zugriffsgeschützt sind. Bei Änderungen der Archive durch den Downloader kann es allerdings vorkommen, dass die Anwendungsblöcke ganz oder teilweise überschrieben werden, um diese Wahrscheinlichkeit zu verringern, werden sie in umgekehrter Richtung gezählt.

4.2.5 Vorverlinken der Java-Klassen

Das Zusammenstellen der Java-Klassen zu Archiven wird von einem in gewöhnlichen Java implementierten *Vorverlinker* auf dem Entwicklungssystem erledigt. Diese Software eignet sich auch für andere Zielsysteme, als die in dieser Arbeit beschriebenen 8-Bit-VM. Die meisten Abläufe lassen sich allgemein anwenden. An dieser Stelle soll nur auf die Abläufe, die hinter der grafischen Bedienoberfläche ablaufen, eingegangen werden. Die 8-Bit-VM kann dahingehend größenoptimiert werden, dass sie nicht mehr in der Lage ist, aus den Klassendateien ihre internen Zugriffstabellen CPR und MRT beim ersten Zugriff auf eine Klasse selbst zu erzeugen. Late Binding ist, wie bereits erwähnt, wegen des knappen flüchtigen Speichers auch wenig sinnvoll. Dann ist sie auf externe Werkzeuge angewiesen, die dies im Voraus für alle Klassen tun (*Early Binding*). Die Aufgaben des Vorverlinkers gehen also darüber hinaus, lediglich Klassen zu einem Archiv zusammenzustellen. Im einzelnen sind zu nennen:

Verwaltung von Projektdateien Eine Klassen- und Ressourcenzusammenstellung für ein Archiv wird verwaltet und kann gespeichert werden.

Zerlegen der Klassen Alle Klassen werden untersucht und in die in [LY00] spezifizierten Datenstrukturen zerlegt.

Konsistenzprüfung eines Projekts Es wird in den zerlegten Klassen geprüft, ob alle referenzierten Klassen im Projekt oder in der VM-internen API-Bibliothek erreichbar sind. Ggf. werden per Suchpfad auffindbare Klassen dem Projekt hinzugefügt. Ist dies nicht möglich, so erfolgt an dieser Stelle ein fehlerhafter Abbruch des Vorverlinkungsvorgangs. Dieser Fehler wird dann aber später nicht zur Laufzeit der Anwendung auf der VM passieren, wo dies u. U. nicht mehr behebbar ist, da ein Nachladen von Klassen zur Laufzeit nicht vorgesehen ist.

Vorverlinken der Klassen Die symbolischen Referenzen auf Klassen, Methoden und Variablen werden zu klassenübergreifenden Objektreferenzen aufgelöst, damit diese mittels Java einfach ansprechbar sind. Die VM-internen Strukturen CPR und MRT werden erzeugt, wobei die durch Vererbung

entstehenden Abhängigkeiten berücksichtigt werden. Ferner werden weitere Hilfsstrukturen angelegt, die am Ende nicht auf dem Zielsystem benötigt werden, z. B. die Vererbungshierarchie von Interfaces oder Klassen- und Instanzvariablen-Tabellen, die die Auflösung von Variablenzugriffen in die entsprechenden Speicherplätze in den Objekten durchnummerieren.

Optimieren der Klassen Die Struktur der CPR ist eng an den Konstantenpool der Klasse gekoppelt, denn alle Referenzen innerhalb einer Klasse erfolgen über Konstantenpool-Einträge. Ein Großteil der Symbole wird nun nicht mehr benötigt, da sie nun komplett durch direkte Java-Referenzen ersetzt wurden, und daher aus der CPR entfernt. Daraus resultiert eine notwendige Umnummerierung der Konstantenpoolreferenzen jeder Klasse. Dazu müssen zusätzlich zu den Datenstrukturen auch die Bytecodes untersucht und modifiziert werden. Ebenfalls entfernt werden für die Ausführung irrelevanten Debug-Informationen der Klassendatei und die damit verbundenen zusätzlichen Symbole.

Erzeugen des Archivs Dieser Vorgang erfordert eine Kommunikation mit der VM bzw. die Kenntnis des Formats eines Archivs, des Formats der umgebenden Datenstrukturen (Segmentliste, Bänke, Flash-Speicherorganisation etc.) und der daraus resultierenden absoluten Zieladresse des Archivs. Es wird ein Abbild des Archivs in der für das Zielgerät erforderlichen Kodierung (i. d. R. binär) erzeugt. Erst an dieser Stelle ist die Verarbeitung zielsystemabhängig und die abstrakten Java-Datensätze werden erstmals in konkrete Datenstrukturen umgesetzt. Auch erst an dieser Stelle entscheidet es sich, ob die oben vorgenommenen Optimierungen für das Zielgerät relevant sind, da auch Geräte, die normale Java-Klassen verarbeiten, denkbar sind. Die Java-Referenzen werden in absolute Zeiger bzgl. der Bank aufgelöst. Die Tabellen mit den übriggebliebenen Symbolen aller Klassen werden zusammengefasst, so dass kein Symbol doppelt vorkommt.

Hochladen des Archivs Schließlich wird dieses Archiv-Abbild auf das Zielgerät geladen. Dabei werden je nach Einstellung vorhandene Archive überschrieben oder ergänzt; die Segmentliste wird bearbeitet. Handelt es sich auf dem Zielgerät um Flash-Speicher, so erfolgt die Übertragung blockweise.

Zusätzlich können Geräteprofile verwaltet bzw. ein Versionsmanagement durchgeführt werden. Diese Notwendigkeit besteht, da die VM alleine kein vollständiges Gerät bestimmt: Es ist zu einem Großteil von der angeschlossenen Hardware abhängig, deren Funktionalität sich in der Laufzeitbibliothek widerspiegelt. Jedem Geräteprofil ist daher eine eigene interne API-Bibliothek

zugeordnet. Ferner existieren externe Bibliotheken, auf die eine Anwendung zusätzlich zugreifen kann. Von dort benötigte Klassen werden zusammen mit der Anwendung in ein Archiv gepackt.

Kapitel 5

Die Speicherverwaltung

Java verfügt über eine eigene Speicherverwaltung, die in der Programmiersprache über das Schlüsselwort `new`, das ein neues Objekt erzeugt, sichtbar ist. Alle Java-Objekte befinden sich in einer Laufzeitspeicherstruktur, dem *Heap*. Das explizite Entfernen von Objekten aus dem Heap ist nicht vorgesehen und wird bekanntermaßen durch eine automatische Speicherbereinigung (Garbage-Collection) durchgeführt, um Speicherlecks zu vermeiden, die bei nicht sorgfältiger Anwendungsprogrammierung entstehen können. Die Speicherbereinigung, wie auch die eigentliche Verwaltung des Heaps, ist Aufgabe der virtuellen Maschine.

Neben dem Heap sind in der Spezifikation [LY00] weitere Speicherbereiche definiert:

- Die virtuellen Programmzähler eines jeden Threads,
- die Stapelspeicher der virtuellen Maschine, ebenfalls einer je Thread,
- der Methodenbereich, welcher den Java-Bytecode enthält,
- ein Laufzeit-Konstantenpool je initialisierter Klasse und
- ggf. **C**-Stacks, also Stapelspeicher für native Methoden.

Diese Speicherbereiche haben bei dieser Implementierung keine explizite Entsprechung. Alle Daten, die einem Thread zugeordnet sind, befinden sich zusammen mit dem Thread-Objekt auf dem Heap. Der Laufzeit-Konstantenpool wird größtenteils nicht benötigt, weil er bereits vorab aufgelöst wurde. Übriggeblieben sind lediglich kleine Tabellen mit Laufzeitreferenzen, die jeweils einem Klassen-Objekt auf dem Heap zugeordnet sind. Der Methodenbereich befindet sich im Festwertspeicher des Controllers. Die Stapel für nativen Code entfallen hier ebenfalls, was noch in Abschnitt 6.3 betrachtet wird. Bei dieser Implementierung der virtuellen Java-Maschine vereinigt der Heap also alle spezifizierten Laufzeit-Speicherbereiche, es handelt sich um einen *Unified Heap*. Das vereinfacht die Speicherverwaltung, eine separate Behandlung verschiedener Bereiche

ist nicht nötig, eine Koordinierung dieser Bereiche untereinander kann ebenfalls entfallen.

5.1 Der Heap

Java geht mit Speicher im höchstem Maße dynamisch um. Das häufige Anlegen von Objekten und deren spätere Verwerfung wird durch die Struktur der objektorientierten Programmiersprache gefördert. Da die Objekte in der Regel nicht gleich groß sind, entstehen zwischen den Objekten freie Bereiche, die nicht immer für neue Objekte groß genug sind, es entsteht ein Verschnitt des Speichers, die sog. *Speicherfragmentierung*. Die virtuelle Maschine muss diese mit geeigneten Verfahren verhindern und ungenutzten Speicher zusammenfassen. Dies ist umso wichtiger, je knapper der zur Verfügung stehende Speicher ist. Ferner sollte sich der Aufwand der Speicherverwaltung in Grenzen halten und möglichst wenig Code benötigen. Schließlich sind auch noch Echtzeitanforderungen zu berücksichtigen: Das Anfordern eines Heap-Blocks sollte in einer vorhersagbaren Zeit stattfinden können. Im Folgenden werden einige grundlegende Verfahren der Speicherverwaltung vorgestellt, die für eingebettete Systeme in Frage kommen:

Handletabelle Um eine Fragmentierung zu vermeiden, können Heap-Blöcke verschoben werden. Damit die Blöcke dennoch unter einer festen Adresse referenziert werden können, wird eine weitere Referenzebene eingefügt, die *Handletabelle*. Die Blöcke werden also nicht direkt über einen Zeiger referenziert, sondern indirekt über einen Index in einer Handletabelle, deren Einträge nicht verschoben werden. Nach der Freigabe von Blöcken werden die Lücken gefüllt, indem Blöcke verschoben werden, was Aufgabe der Speicherbereinigung ist. In der Folge existiert immer ein freier Bereich, auf den direkt ein Zeiger verweist, der für neue Blöcke verwendet wird (*Nursery*). Das Anlegen von Blöcken kann also in einer vorhersagbaren Zeit erfolgen. Natürlich kostet das Verschieben der Blöcke Zeit, so dass unter Umständen der freie Bereich bei einer Heap-Anforderung noch nicht zur Verfügung steht. Ein weiterer Nachteil dieses Verfahrens ist der etwas höhere Heap-Verbrauch, da je Block ein zusätzliches Wort in der Handletabelle benötigt wird. Dies relativiert sich jedoch bei der Implementierung der Speicherbereinigung, die keine zusätzlichen Listen benötigt.

Direkte Referenzen Wird auf die Handletabelle verzichtet und werden direkte Referenzen verwendet, so sind die Zugriffe auf die Heap-Blöcke performanter. Soll eine Kompaktierung des Heaps erfolgen (Blöcke werden

verschoben), so ist ein größerer Aufwand zu betreiben, um die Zeiger in allen referenzierenden Blöcken anzupassen. Dieser Vorgang ist nicht unterbrechbar (bei dem Verfahren mit der Handletabelle ist nur das Verschieben des Blocks selbst und die Anpassung des Eintrags in der Handletabelle nicht unterbrechbar). Optimierungen der Speicherbereinigung sind nötig für eine ausreichende Performance (z. B. *Generational Garbage-Collection*) [5].

Feste Blockgrößen Üblicherweise haben Blöcke auf dem Heap unterschiedliche Größen, was allein zu dem Fragmentierungsproblem führt. Wird die Blockgröße also für alle Blöcke festgeschrieben, so können die Kompaktierung und alle damit verbundenen Probleme entfallen. Da die Blöcke nicht verschoben werden, können sie mit direkten Zeigern referenziert werden. Dieses Verfahren ermöglicht sogar eine Speicherbereinigung unter harten Echtzeitbedingungen [Sie02]. Der Nachteil hier ist der aufwändigere Zugriff auf einen Block, da größere Blöcke auf mehrere gleich kleine Einzelblöcke aufgeteilt werden müssen. Es sind also immer Listen- und Baumstrukturen zu verfolgen. Ferner gibt es für jeden zusammengesetzten Block einen durchschnittlichen Speicherverschnitt einer halben Blockgröße.

Das Verfahren mit der Handletabelle lässt sich mit dem geringsten Aufwand und dem geringsten Code-Umfang implementieren und wird bei der Yogi2-VM verwendet.

5.1.1 Die Struktur des Heaps

Beim Zielsystem wird ein zusammenhängender Speicherbereich für den Heap benutzt, der bei niedrigen Adressen die Datenblöcke enthält und bei hohen Adressen die Handletabelle. Bei der Allozierung von Blöcken wachsen beide Bereiche aufeinander zu, der Zwischenraum wird für die neuen Blöcke verwendet (Nursery). Die Handletabelle enthält die Zeiger auf die Blöcke (16 Bit). Einige ungenutzte Bits werden von der Speicherbereinigung verwendet (siehe Abschnitt 5.2). Die Zeiger weisen direkt auf die Datenbereiche der Blöcke, die die Java-Objekte aufnehmen. Jeder Block verfügt über einen Verwaltungsbereich, der sich vor dem Datenbereich befindet und über den Zeiger mit negativer Indizierung erreichbar ist. Er enthält einen Block-Identifier (4 Bit), der den Block-Typ bestimmt, sowie die Block-Größenangabe (4 Bit bis 16 Bit variabel). Bild 5.1 verdeutlicht diesen Aufbau.

Die JavaVM kann jeden Block eindeutig einem Typ zuordnen und dann auf die entsprechenden Datenstrukturen zugreifen. Es wird grob zwischen Blö-

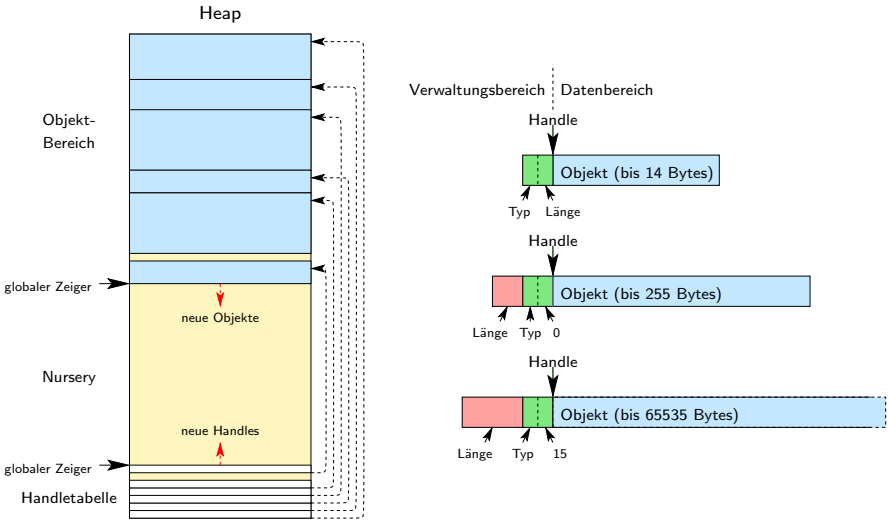


Bild 5.1: Die Struktur des Heaps

cken, die Java-Objekte enthalten, und VM-internen Blöcken unterschieden, auf letztere hat ein Java-Programmierer keinen Zugriff. Tabelle 5.1 stellt die Typen zusammen und nennt die verwendeten Datensätze. Blöcke, die dem Typ `Objekt` angehören, verfügen über einen Header mit der Referenz auf den Typ (die Klasse), dem das Objekt angehört, sowie Synchronisierungsinformationen (siehe Abschnitt 7.1.2). Insgesamt ist der Header sechs Bytes groß. Blöcke vom Typ `Array` verfügen nur über diesen Header, alle übrigen Objekte, bei denen es sich sich um Instanzierungen von Klassen handelt, zusätzlich über den Satz der Exemplarvariablen des Typs (Datenfeld). Eine Ausnahme bilden dabei die Objekte vom Typ `Class`, dort werden an dieser Stelle die statischen (klassenbezogenen) Variablen abgelegt. Die bereits in Abschnitt 4.2.3.3 beschriebenen Laufzeitklassenreferenzen befinden sich ebenfalls dort.

Es gibt Objekte mit einer speziellen Erweiterung, auf die nur die JavaVM oder dedizierter nativer Code der Klasse direkt zugreifen kann. Hierbei handelt es sich um eine festgelegte Schnittstelle zwischen VM und dem Modul, das diesen Objekttyp verwaltet (das kann auch Code innerhalb der VM-Laufzeitumgebung sein). Diese Kommunikation wäre zwar auch über das Datenfeld des Objekts möglich, allerdings ist u. U. das Datenfeld nicht in der Reihenfolge der Einträge festgelegt (abhängig vom verwendeteten Java-Compiler), der Implemen-

Tabelle 5.1: Heap-Blöcke

Heap-Block	Objekt	Zusatzinformationen
unspecified	–	unspezifizierte Daten
`Bools`	nur Header	Boolean-Array (1 Bit)
`Bytes`	nur Header	Byte-Array (8 Bit)
`Words`	nur Header	Word-Array (16 Bit)
`References`	nur Header	Objekt-Array (16 Bit)
`Init`	–	Klassenlader-Datensatz
`Object`	Variablen	–
`Class`	statische Variablen	Laufzeitdatensatz der Klasse
`Thread`	Variablen	Thread-Datensatz
`String`	Variablen	Zeichenkette oder Zeiger
`Deadline`	Variablen	Scheduling-Datensatz

tierungs- und Verwaltungsaufwand für die Zugriffe der VM wären höher. Die genaue Zusammensetzung dieser Erweiterungsstrukturen kann in Anhang C.2 nachgeschlagen werden und soll an dieser Stelle nicht näher beschrieben werden. Auf die Daten eines Blocks vom Typ Thread und die Funktionsweise der Threads selbst wird im Kapitel 7 noch näher eingegangen.

5.2 Speicherbereinigung

Die automatische *Speicherbereinigung* (Garbage-Collection) ist ein fundamentales Element einer JavaVM und in der Spezifikation vorgeschrieben. Bei großen Java-Systemen hat sich die Speicherbereinigung oft als Flaschenhals herausgestellt. Erst aufwändige Algorithmen ermöglichen eine Bereinigung ohne allzu großen zeitlichen Overhead. Üblich ist heutzutage ein hierarchischer Ansatz, der sich zu Nutze macht, dass es unterschiedliche Nutzungsklassen von Objekten gibt, langlebige und kurzlebige (*„young objects die young“*) [21]. Bei kleinen Systemen spielen diese Optimierungen eine untergeordnete Rolle, da die Anzahl der Objekte alleine durch die geringe Größe des Speichers begrenzt ist.[11] Ziel muss auch hier ein kompakter Algorithmus sein. Ferner ist bedingt durch die knappen Ressourcen ein speicherplatzschonendes Verfahren wichtig. Zusätzliche Zähler oder Listen sind zu vermeiden.

Als Grundlage für die Wahl eines geeigneten Speicherbereinigungs-Algorithmus dient [Wil92], das aus der Zeit stammt, bevor die JavaVM erfunden wurde,

[11] Auf dem später noch vorgestellten Zielsystem sind z. B. maximal 268 Objekte möglich, in praktischen Anwendungen in der Regel weniger als 100.

aber dennoch einen guten Überblick über die Speicherbereinigung und deren Geschichte bietet. Die vollständige Beschreibung der hier erwähnten und nur kurz skizzierten Algorithmen befindet sich dort.

5.2.1 Erkennen von Speichermüll

Eine Speicherbereinigung kann auf unterschiedliche Weise nicht mehr benötigte Blöcke auf dem Heap erkennen. Diesbezüglich werden nun einige Verfahren betrachtet Es können grundsätzlich implizite und explizite Verfahren unterschieden werden. Ein implizites Verfahren ist:

Reference Counting Jeder Speicherblock erhält einen Zähler, der die Anzahl der Referenzen auf diesen Block enthält. Dieses Verfahren ermöglicht eine exakte Speicherbereinigung mit der Ausnahme, dass zirkulare Abhängigkeiten auftreten können, die nicht erkannt und bereinigt werden. Hauptnachteil dieses Verfahrens ist die Konsistenzhaltung des Zählerstandes, die bei jeder Änderung einer Referenz erfolgen muss. Ferner belegt auch der Zähler Heap-Speicherplatz. Damit Reference Counting richtig funktioniert, müssen Referenzen eindeutig von anderen Werten unterschieden werden können. Dazu müssen in jedem Block, der Referenzen enthalten kann, u.U. Flag-Felder eingeführt werden. Reference Counting kann für direkte und indirekte Referenzen angewendet werden. Sobald der Zähler eines Blocks auf Null geht, kann der Block sofort entfernt werden, was direkt im referenzverändernden Code, dem *Mutator*, geschieht und nicht in der Speicherbereinigung. Eine Fragmentierung tritt dennoch auf, so dass regelmäßig eine Kompaktierung erfolgen muss.

Aufgrund des hohen Aufwands beim Schreibzugriff auf Referenzen und dem erhöhten Speicherbedarf ist Reference Counting weniger für kleine Heaps geeignet. Explizite Verfahren zur Erkennung von Speichermüll verzichten hingegen weitestgehend auf aufwändigen Code in den Mutatoren, stattdessen findet der Hauptteil der Untersuchungen in der Speicherbereinigung selbst statt:

Listen Die Blöcke werden in Gruppen (Farben) unterteilt. Jede Gruppe findet in einer verketteten Liste Platz, die die Speicherbereinigung verwaltet. Zu Beginn der Markierungsphase befinden sich alle Böcke in der Gruppe *weiß*, bis auf bestimmte zu erhaltende Blöcke (Root Set), die in der Gruppe *grau* sind. Schließlich werden alle grauen Blöcke untersucht und die darin referenzierten Blöcke ebenfalls in die graue Gruppe aufgenommen. Fertig untersuchte Blöcke werden in die Gruppe *schwarz* überführt. Bei diesem Verfahren muss sichergestellt werden, dass in den fertig untersuchten Blöcken (schwarz) niemals Referenzen auf noch nicht untersuchte

Blöcke (weiß) stehen können (*Färbungs-Invariante*). Am Ende existieren nur noch Blöcke in den weißen und schwarzen Listen, letztere bleiben erhalten und erstere werden in freien Speicherplatz umgewandelt. Ein eleganter Algorithmus hierfür ist Bakers Treadmill [Wil92], welcher eine doppelt verkettete Liste, die in Segmente eingeteilt ist, für alle Blöcke verwendet. Folglich sind je Block zwei Referenzen zusätzlich nötig.

Markierungen Es wird auf die Verwaltung von Listen verzichtet und stattdessen die Einteilung der Blöcke in die Gruppen *weiß*, *grau* und *schwarz* durch Markierungen in den Köpfen der Blöcke oder in den entsprechenden Einträgen der Handletabelle vorgenommen. Dazu reichen zwei Bit pro Block aus, die u. U. sogar in ungenutzten Bits bereits vorhandener Daten untergebracht werden können. Hauptnachteil dieses Verfahrens ist, dass die Blöcke nicht nach Gruppen sortiert vorliegen und die Speicherbereinigung den Heap immer wieder nach grauen Blöcken absuchen muss.

Bei kleinen Heaps wird dem *Markierungsverfahren* wegen des geringen Speicherverbrauchs der Vorzug gegeben. Der Suchaufwand fällt dort wegen der geringen maximalen Blockzahl weniger ins Gewicht. Bei größeren Heaps ist hingegen die Verwendung von Listen von Vorteil, hier überwiegt der Zeitgewinn, da in der Markierungsphase die Blöcke nur einmal durchlaufen werden müssen, gegenüber dem Mehrverbrauch von Heap-Speicherplatz. Bei beiden expliziten Verfahren ist es nicht nötig, dass alle Referenzen von anderen Werten unterschieden werden müssen. Zusätzliche Flag-Felder werden also nicht benötigt. Dabei können allerdings Werte fälschlicherweise als Referenzen erkannt werden und einige Blöcke werden nicht als Speichermüll erkannt und bleiben erhalten. Die Speicherbereinigung hat eine erhöhte *Konservativität* und ist nicht exakt.

5.2.2 Speicherdefragmentierung

Um die Speicherfragmentierung zu behandeln, kommen zwei Verfahren zum Einsatz:

Copy Die grau (und schwarz) markierten Blöcke werden in einen zweiten Heap-Bereich kopiert (To-Space). Dabei verschwinden die nicht referenzierten (weißen) Blöcke. Dieses Verfahren hat natürlich den Nachteil, dass mehr Speicher benötigt wird. Um dem entgegenzuwirken, kann der Heap nicht in zwei, sondern in mehrere, kleinere Segmente unterteilt werden. Die Bereiche werden dann nacheinander bereinigt. Die Größe der Heap-Segmente begrenzt allerdings die maximale Blockgröße, was insbesondere bei kleinen Heaps das Optimierungspotenzial begrenzt.

Compact Die Blöcke werden im selben Speicherbereich zusammengeschoben (kompaktiert). Hierbei tritt kein Speicherverschnitt auf. Auch dieses Verfahren hat einen Nachteil, im Gegensatz zum Kopieren in einen anderen Heap, müssen die Blöcke in der richtigen Reihenfolge (von oben nach unten) verschoben werden, hierbei fällt entweder wieder ein Suchaufwand an, oder es müssen die Datenstrukturen entsprechend angepasst werden, beispielsweise könnte jeder Block eine Referenz auf den nächsten Block enthalten. Diese beiden Verfahren werden in Abschnitt 5.3 noch genauer betrachtet.

Bei kleineren Heaps ist trotz des möglichen Suchaufwands oder erhöhtem Speicherbedarf die Verwendung eines *Mark-Compact* Garbage-Collectors mit dem Dreifarb-Markierungsverfahren vorzuziehen. Im Extremfall sind lediglich die Farbmarkierungsflags hinzuzufügen, diese finden zusammen mit der Adresse eines Blocks in dem Wort der Handletabelle Platz.

Bei größeren Heaps sollte entweder ein *Mark-Copy* Garbage-Collector verwendet werden (welcher einfacher zu implementieren ist) oder zumindest durch geeignete Datenstrukturen der Suchaufwand beim Kompaktieren vermieden werden. Ein Mark-Compact Garbage-Collector hat auch bei größeren Heaps den Vorteil, dass der Kopieraufwand beim Verschieben von Blöcken verringert wird. Es müssen nicht bei jedem Durchlauf alle Blöcke umkopiert werden, ggf. bleiben langlebige Objekte bei niedrigen Adressen liegen und es entstehen dort für längere Zeit keine Freigaben.

5.2.3 Nebenläufige Speicherbereinigung

Viele Speicherbereinigungen müssen die laufende Anwendung anhalten, um ausgeführt zu werden (die sogenannte Stop-the-World Garbage-Collection). Stattdessen kommt bei der Yogi2-VM eine *nebenläufige* Speicherbereinigung zum Einsatz. Sie läuft in einem eigenständigen Programmfaden (Thread) unabhängig von der Anwendung ab. Bereits in [DKL+00] wurde gezeigt, dass diese Vorgehensweise auf Server-Systemen mit vielen Threads Vorteile bietet. Die zugrundeliegende Idee dahinter ist, dass nicht die Speicherbereinigung entscheiden soll, wann sie läuft, sondern der Scheduler des Echtzeit-Kerns (siehe Kapitel 7). Nur der Scheduler kann bewerten, wann es möglich oder sinnvoll ist, zur Speicherbereinigung zu wechseln, oder ob es gegenwärtig andere Anforderungen im Gesamtsystem gibt, wenn z. B. ein Thread mit Echtzeitanforderungen vorhanden ist.[12] Damit die Speicherbereinigung sinnvoll in einem eigenen Thread

[12] Die Speicherbereinigung ist in dieser Implementierung ein Teil des System-Threads (siehe Abschnitt 8.1.1), weitere Anmerkungen zum Echtzeitverhalten der Speicherbereinigung finden sich dort.

arbeitet, wird die Ausführung in möglichst viele und kleine Bearbeitungsschritte unterteilt (die nebenläufige ist ein Sonderfall der inkrementellen Speicherbereinigung). Der grobe Ablauf wird dabei in Phasen eingeteilt, von denen jede die Speicherblöcke in einer Schleife abarbeitet. Jeder Bearbeitungsschritt ist dabei in sich abgeschlossen und spiegelt sich in einer eindeutigen Zustandsänderung eines Blocks wider.

Die Abstraktion des Speicherbereinigungs-Zustands eines jeden Speicherblocks mit Farbmarkierungen ist ein Ansatz, der es grundsätzlich erlaubt eine nebenläufige oder inkrementelle Speicherbereinigung durchzuführen. Bei einer nebenläufigen Speicherbereinigung existiert allerdings eine Möglichkeit, die Regel zu verletzen, dass ein schwarz gefärbtes Objekt niemals ein weiß markiertes enthalten darf (Färbungs-Invariante). Das Problem tritt immer dann auf, wenn eine Anwendung (der Mutator) während der Markierungsphase der Speicherbereinigung Änderungen an bereits untersuchten Blöcken vornimmt. Theoretisch müsste also bei allen Zugriffen geprüft werden, ob sie beim Verändern von Referenzen diese Regel verletzen. Es gibt grundsätzliche Verfahren, die Mutatoren mit der Speicherbereinigung zu koordinieren [Wil92]:

1. Eine *Read-Barrier* verhindert, dass ein Mutator Referenzen auf weiß gefärbte Objekte lesen kann, indem sie bei Lesezugriffen grau gefärbt werden. Dies wird durch entsprechendes Umkopieren in den To-Space, Umsortieren der Listen oder durch Setzen der Markierungen erledigt. Somit können keine Referenzen auf weiße Objekte in schwarzen installiert werden. Dieses Verfahren ist ob der häufigen Lesezugriffe recht teuer, funktioniert aber bei allen Speicherbereinigungsverfahren uneingeschränkt.

2. *Write-Barrier*-Ansätze sind etwas direkter und verhindern das Schreiben von nicht erlaubten Referenzen:

   a) Eine Möglichkeit ist, beim Überschreiben von Referenzen den vorherigen Wert grau zu färben, diesen also auf jeden Fall von der Speicherbereinigung zu untersuchen (*Snapshot-at-Beginning*). Auf diese Weise können keine Referenzen übersehen werden auch dann, wenn die überschriebenen Referenzen in bereits schwarz markierten Blöcken gesichert werden. Da die Blöcke auch erhalten bleiben, wenn sie nicht woanders gesichert wurden, also Speichermüll sind, ist die Konservativität bei diesem Verfahren sehr hoch.

   b) *Incremental-Update*-Verfahren verhindern das Installieren von weißen Referenzen in schwarzen Blöcken, indem entweder der schwarze Block wieder grau gefärbt wird oder der referenzierte weiße Block. Im ersten Fall wird die Speicherbereinigung den schwarzen Block erneut untersuchen und ggf. den weißen Block markieren, falls die

Referenz auf ihn dann immer noch existiert. Im zweiten Fall wird der Block in jedem Fall markiert, die Konservativität ist also höher.

Da Schreibzugriffe auf Referenzen seltener sind, als Lesezugriffe, sind die Write-Barrier Ansätze hier im Vorteil. Der Incremental-Update-Ansatz mit der direkten Markierung der referenzierten Blöcke liefert dabei einen guten Kompromiss aus Rechenzeitbedarf (sowohl für Mutatoren und Speicherbereinigung) und Konservativität und wird daher bei der Yogi2-VM eingesetzt.

Schließlich stellt sich die Frage, wie häufig solch ein Fall auftritt und der Schreibzugriff von Referenzen dadurch verlangsamt wird. Im Kapitel 6 werden noch einige Eigenschaften bei der Ausführung von Bytecode definiert. Es gibt dabei atomare Codesequenzen, die auf jeden Fall zusammenhängend ausgeführt werden. Das betrifft sowohl die Ausführung von Bytecodes (die Mutatoren) als auch die Speicherbereinigung selbst: Ein grau gefärbter Block wird in einem Durchgang nach Referenzen untersucht, die referenzierten Blöcke grau gefärbt und der untersuchte Block anschließend schwarz. Es bleiben also nur sehr wenige Mutatoren übrig, die tatsächlich die Färbungs-Invariante verletzen können, z. B. fallen alle Mutatoren heraus, die lesend und schreibend auf denselben Block wirken.

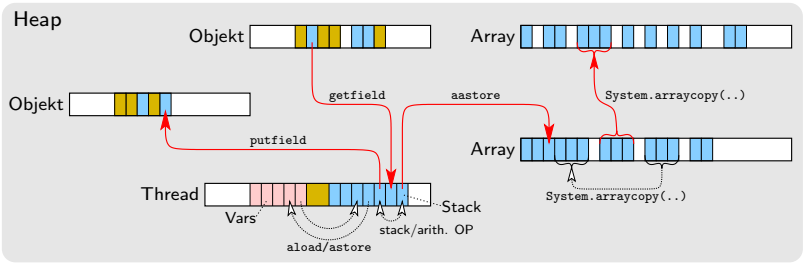


Bild 5.2: Mutatoren und die Färbungs-Invariante

Bild 5.2 zeigt einige Java-Mutatoren auf, die die Färbungs-Invariante verletzen können. Es zeigt sich, dass nur wenige Bytecodes (`getfield`, `putfield`, `getstatic`, `putstatic`, `aaload` und `aastore`) sowie eine Methode der Java-Laufzeitbibliothek (`System.arraycopy(..)`) betroffen sind. Die meisten Bytecodes wirken lediglich auf den Stack desselben Threads, auch der Zugriff auf lokale Variablen findet innerhalb einer Thread-Struktur (siehe Abschnitt 6.2) statt, ebenso die Parameterübergabe beim Methodenaufruf. Array-Zugriffe verfügen über eine explizite Typangabe, so dass die Bytecodes für Zugriffe auf numerische Arrays nicht geprüft werden müssen. Bei Zugriffen auf Objekt- und

Klassen-Attribute kann hingegen nur implizit über den referenzierten Konstantenpool-Eintrag auf den Typ geschlossen werden, diese Typisierung kann durch die in Abschnitt 4.2.3.1 erwähnten Optimierungen entfallen, so dass hier ein konservativer Ansatz nötig ist.

5.3 Rechenzeitverbrauch

Ein weiterer Faktor, der bei einer nebenläufigen Speicherbereinigung betrachtet werden muss, ist die verwendete bzw. zugeteilte Rechenzeit. Wird der Speicherbereinigung zu wenig Rechenzeit zugeteilt, so dass sie langsamer erfolgt, als für neu zu erzeugende Objekte nötig, so wird der Heap-Speicher volllaufen. Bei einer Speicheranforderung muss dann der entsprechende Anwendungs-Thread warten, bis die Speicherbereinigung ihre Kompaktierungsphase beendet hat. Insbesondere wenn hierbei Echtzeitanforderungen eine Rolle spielen, sollte dies vermieden werden. Die Rate, mit der die nebenläufige Speicherbereinigung aufgerufen wird, sollte daher proportional zu der aktuellen Speicherallokationsrate sein und umgekehrt proportional zu der Größe des noch freien Heaps (Nursery) [Nil98]. Die Ausführungsrate kann dabei auch Null werden, wenn keine Allokationen auftreten. Das ist Voraussetzung, um den Controller des Zielsystems in Bearbeitungspausen in einen Energiesparmodus versetzen zu können.

Auf dem Zielsystem wurden einige Messungen der Ausführungszeiten der Speicherbereinigung im Verhältnis zu denen von Anwendungen durchgeführt. Für diesen Test wurde die virtuelle Maschine um eine Messvorrichtung für die CPU-Zeit eines jeden einzelnen Threads erweitert.[13] Auf der Anwendungsebene lassen sich die Zähler auslesen und zurücksetzen. In dem Thread, der für die Speicherbereinigung zuständig ist, wird die Rechenzeit zusätzlich entsprechend der Bearbeitungsphasen erfasst. Eine vollständige Übersicht der Testprogramme und Messergebnisse findet sich in Anhang E.3.

Um Messungen an der Speicherbereinigung durchzuführen, wurde eine Testanwendung verwendet, die eine Baumstruktur von abhängigen Objekten erzeugt und anschließend mit konstanter Rate Objektreferenzen löscht (also die Objekte für die Speicherbereinigung vorsieht) und neue Objekte anlegt. Das Flächendiagramm in Bild 5.3 zeigt den Rechenzeitverbrauch der Anwendung, sowie der Markierungs- und Kompaktierungsphasen der Speicherbereinigung bei einer konstanten Allokationsrate und variabler Anzahl auf dem Heap residenter Objekte. Die Messung wurde für fünf Sekunden angesetzt, bei den Messreihen, die insgesamt weniger Rechenzeit benötigen, konnte der Controller

[13] Dazu wurde der VM-interne Millisekundentimer verwendet, der auch den Scheduler steuert, siehe Abschnitt 7.1.1.

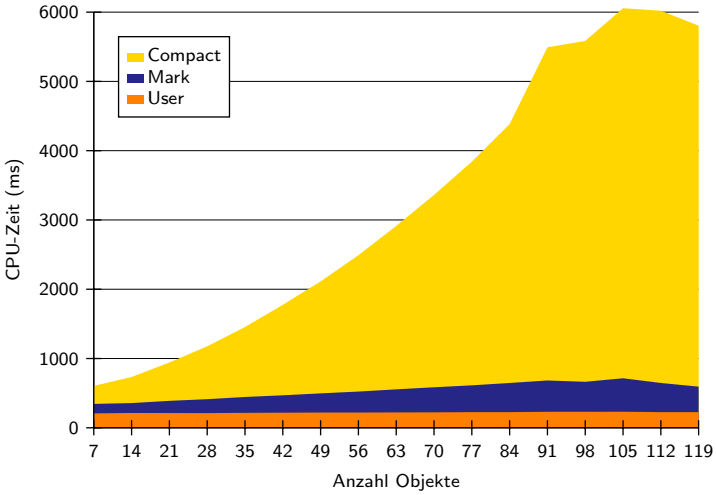


Bild 5.3: Rechenzeitverbrauch der Speicherbereinigung bei konstanter Allokationsrate von 500 ms

zwischen den einzelnen mit einer Periode von 500 ms stattfindenden Aktionen in den Ruhezustand versetzt werden. Die Werte schließen je eine vollständige Speicherbereinigung vor und nach den Messungen ein, so dass die vorgesehene Gesamtzeit überschritten werden kann. Zu erkennen ist in Abhängigkeit von der Speicherbelegung (Anzahl der residenten Objekte) ein konstanter Rechenzeitverbrauch der Anwendung und eine lineare Abhängigkeit der Markierungsphase der Speicherbereinigung. Aufgrund des nötigen Suchvorgangs je Speicherblock ergibt sich für die Kompaktierung des Heaps eine quadratische Abhängigkeit von der Speicherbelegung.

Diese Messungen wurden so ausgelegt, dass für jede Allokation ein vollständiger Durchlauf der Speicherbereinigung erfasst wird. In praktischen Anwendungen ist dies natürlich ein seltener Fall, dort kommen Allokationen unabhängig vom Zustand der Speicherbereinigung vor. Bei einer höheren Speicherverwendung wird die benötigte Rechenzeit der Speicherbereinigung so groß, dass vor der nächsten Speicheranforderung keine vollständige Bereinigung durchgeführt werden kann, dann reduziert sich die Anzahl der Speicherbereinigungs-Zyklen, die Anwendung bleibt davon unberührt. In Bild 5.3 ist das beispielsweise ab einer Zahl von 98 residenten Objekten der Fall.

Das Anlegen neuer Objekte scheitert dann, wenn die Speicherbereinigung die

freigegebenen Objekte nicht rechtzeitig freigeben kann. Dieser Fall ist in Bild 5.4 zu sehen. Hier läuft derselbe Test mit einer verringerten Allokations- und Freigabeperiode von 100 ms. Ab einer Speicherbelegung von 91 residenten Objekten können die Echtzeitvorgaben der Anwendung nicht mehr eingehalten werden und es müssen Allokationen ausgelassen werden.[14] Im Endeffekt führt dies zu einer verringerten aufgewendeten Rechenzeit für die Anwendung (Performance-Verlust).

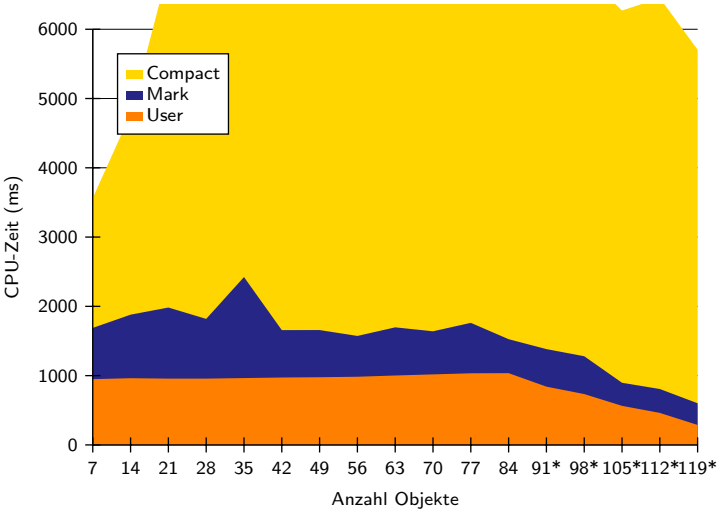


Bild 5.4: Rechenzeitverbrauch der Speicherbereinigung bei konstanter Allokationsrate von 100 ms

5.3.1 Alternative Speicherverwaltung

Auf einigen Zielsystemen können Anforderungen an die Speicherverwaltung und das Echtzeitverhalten des Systems bestehen, die von einer solchen wenig effektiven Speicherbereinigung nicht mehr getragen werden können (obwohl es generell wenig sinnvoll ist, bei Echtzeit-Aufgaben Speicherallozierungen durchzuführen).

[14] Die Verletzung der Echtzeitvorgaben löst in dem Testprogramm eine `DeadlineMiss Exception` aus, die in der Anwendung ein Fehler-Flag setzt, das in Bild 5.4 durch Sternchen markiert wird. Eine genaue Beschreibung der Mechanismen zur Echtzeitverarbeitung findet sich in Abschnitt 7.2.

Weiterhin sorgt eine ineffektive Speicherbereinigung für einen erhöhten Energiebedarf. Als Schwachpunkt hat sich in diesen Messungen die Kompaktierung des Heaps heraus gestellt, da die sequentielle Suche nach dem nächsthöheren Block für jeden Block erfolgen muss und in einer Komplexität von $O(n^2)$ mit n, der Anzahl der Blöcke auf dem Heap, resultiert.[15] Bei einer nur leicht aufwändigeren Implementierung der Speicherverwaltung, die eine einfach verkettete nach Adressen sortierte Liste für die Blöcke verwendet, kann die Suche nach dem nächsten zu verschiebenden Block entfallen und die Komplexität der Kompaktierung sinkt auf $O(n)$,[16] siehe Bild 5.5.

Auch für Speicher- und Rechenintensive Anwendungen ohne Echtzeitvorgaben bringt der optimierte Kompaktierer Vorteile bei der Verarbeitungsgeschwindigkeit. Bei den Messungen in Bild 5.6 wurde wieder derselbe Testrahmen verwendet, es wurden lediglich die Echtzeitvorgaben entfernt und die Allokationen mit der höchstmöglichen Geschwindigkeit durchgeführt. Der Graph zeigt die erreichten Iterationen im Verhältnis zur Speicherbelegung. Die Verwendung einer effektiven Speicherbereinigung zeigt auch in realen Anwendungen eine höhere Reaktivität, z. B. wenn viele String-Manipulationen durchgeführt werden (da Strings bei Java nicht veränderbar sind, führt jede Änderung an einer Zeichenkette zu einem neuen Objekt). Auch speicherintensive Anwendungen mit grafischer Bedienoberfläche hinterlassen beim Benutzer einen subjektiv etwas schnelleren Bildaufbau, vergleichende Messungen auf zwei Geräten mit den verschiedenen Speicherverwaltungen bestätigen dies.

[15] Ein weiterer Komplexitätsfaktor der Kompaktierung ist natürlich der eigentliche Aufwand beim Umkopieren der Speicherblöcke, dieser ist begrenzt durch die Größe des verwendeten Heaps (n Blöcke zu m Bytes). In diesen Messungen entspricht dies $O(n)$.

[16] Bei dieser Implementierung wurde die Größenangabe im Verwaltungsbereich der Blöcke durch den Handle des nächsten Blocks ersetzt, die Blockgröße wird nun aus der Differenz der Block-Adressen errechnet. Damit die Freigabe von Blöcken keine Lücken in der Liste hervorruft, werden die Blöcke zunächst nur als frei markiert, der Handle mit dem Zeiger auf den Block bleibt bis zur Kompaktierung erhalten. Bei der Kompaktierung finden zwei Operationen statt: die Vereinigung hintereinander liegender freier Blöcke und das Vertauschen eines freien mit einem belegten Block bei entsprechender Anpassung der Liste.

Dieser Algorithmus benötigt etwas mehr Code auf dem Zielsystem (ca. 180 Byte auf der Referenzimplementierung im Vergleich zu der einfachen Implementierung mit der sequentiellen Blocksuche).

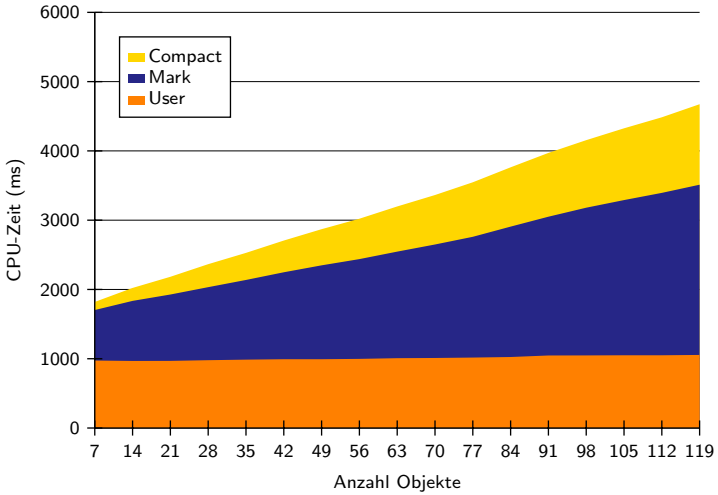


Bild 5.5: Rechenzeitverbrauch der Speicherbereinigung bei konstanter Allokationsrate von 100 ms bei Verwendung von verketteten Listen zur Blockverwaltung

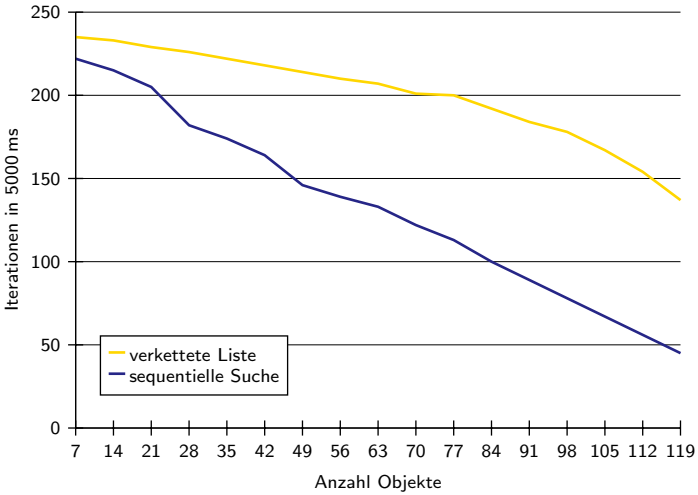


Bild 5.6: Verarbeitungsgeschwindigkeit bei der Allokation und Freigabe von Blöcken bei unterschiedlichen Implementierungen der Speicherverwaltung

Kapitel 6

Ausführen von Java-Bytecode

Wie schon mehrfach erwähnt, wird der Java-Bytecode direkt aus dem Festwertspeicher ausgeführt bzw. interpretiert. Als Tribut an den geringen Speicherplatzverbrauch finden keine Geschwindigkeitsoptimierungen durch Compilertechniken statt. Die Interpretierung von Bytecode gleicht der Vorgehensweise eines jeden (auch realen Prozessors) bei der Ausführung von Programmcode [HP94], nur mit dem Unterschied, dass der Vorgang in Software gelöst ist:

IF – Instruction Fetch Es existiert eine Variable, die einen Java-Programmzähler (*JPC*) enthält; dieser weist an die Stelle (Adresse) der internen oder externen Speicherbank, wo sich der als nächstes auszuführende Bytecode befindet. Der Bytecode wird ausgelesen.

ID – Instruction Decode Der Wert des Bytecodes [LY00] wird benutzt, um eine Sprungtabelle zu adressieren, die die direkten Zieladressen der bytecodeimplementierenden Routinen enthält. Einige ähnliche Bytecodes konnten zusammengefasst werden und weisen auf dasselbe Ziel. Nicht implementierte Bytecodes (dies resultiert aus den Einschränkungen der VM, siehe Abschnitt 4.2.2) führen zu einer Routine, die einen Fehler auslöst. Zur Ausführung des Bytecodes wird die Routine angesprungen.

EX – Execution Die Ausführung der Bytecodes wird in individuellen Routinen kodiert, die den weiteren Ablauf (also auch die beiden folgenden Punkte) übernehmen. Im Gegensatz zu den meisten realen Maschinen fallen einige Bytecodes bei der virtuellen Java Maschine sehr komplex aus, als Beispiel seien Methodenaufrufe genannt, die der Objektorientierung der Sprache Java Rechnung tragen.

MEM – Memory Access Während der Ausführung kann auf Operanden zugegriffen werden, die im Ausführungskontext liegen (dieser wird in Abschnitt 6.2 noch genauer erläutert).

WB – Write Back Die Ergebnisse der Ausführung (falls vorhanden) werden in den Ausführungskontext gesichert.

Nach der Ausführung eines Bytecodes wird der `JPC` um die benötigen Bytes hochgezählt (mit dem Bytecode können direkte Operanden verknüpft sein) und an den Beginn dieser Hauptschleife der virtuellen Maschine gesprungen. Dort wird zunächst eine Abbruchbedingung in einer Variable geprüft, ist diese nicht erfüllt (der Wert `0`), so wird normal mit der nächsten IF-Phase fortgefahren.

6.1 Bytecode-Interrupts

Die Abbruchbedingung des Bytecode-Interpreters ist als Bit-Feld kodiert, von dem jedes Flag einem *Bytecode-Interrupt* entspricht. Der wichtigste Bytecode-Interrupt ist das *Schedule-Flag*, das den Bytecode-Interpreter auffordert, sich kontrolliert zu beenden und zum Aufrufer zurückzukehren, dort findet das Thread-Scheduling statt (siehe Kapitel 7).[17] Der Bytecode-Interpreter sichert seinen Zustand in den Ausführungskontext, so dass später eine Fortsetzung der Ausführung möglich ist. Weitere Bytecode-Interrupts werden von der Laufzeitbibliothek verwendet, um auf Hardware-Ereignisse reagieren zu können (siehe Abschnitt 8.2). In diesem Fall wird nicht zum Scheduler zurückgekehrt, sondern eine für dieses Ereignis registrierte native Routine angesprungen, die die Verarbeitung des Ereignisses zwischen der Bearbeitung der Bytecodes übernimmt. Dieser Mechanismus gewährleistet, dass sich die VM in einem definierten Zustand befindet, und somit von dieser Routine auch auf die Datenstrukturen der VM zugegriffen werden kann. Das ganze Konzept der Bytecode-Interrupts ist analog zu den bei Mikroprozessoren und -controllern verwendeten nativen Interrupts zu verstehen, die direkt auf Hardware-Ereignisse reagieren. Die übliche Vorgehensweise ist, wenn auf beliebige VM-Datenstrukturen auf dem Heap zugegriffen werden muss, in einem nativen Interrupt das Flag des entsprechenden Bytecode-Interrupts zu setzen, wo die Bearbeitung des Ereignisses später fortgesetzt wird. Typische Fälle hierfür sind das Aufwecken von Java-Threads, die auf ein externes Ereignis warten, und der Zugriff auf Ringpuffer auf dem Java-Heap.

[17] Das hier verwendete Grundprinzip des *atomaren Bytecodes*, d. h. dass ein Kontextwechsel immer an Bytecode-Genzen erfolgt, sorgt dafür, dass sich jeder Thread immer in einem definierten Zustand befindet, wenn er von außerhalb betrachtet wird. Dies vereinfacht einiges bei der von der Spezifikation geforderten Datenkonsistenz, der dort ein eigenes Kapitel gewidmet ist [LY00]. Im Endeffekt bedeutet dies, dass alle Java-Variablen sich so verhalten, als seien sie mit dem Java-Schlüsselwort *`volatile`* deklariert worden. Eine Ausnahme bilden dabei lediglich native Interrupts.

6.2 Ausführungskontext

Zur Ausführung von Bytecodes einer Java-Methode ist zum einen die Referenz auf die Klasse, in der die Methode implementiert ist, nötig (statischer Kontext) und zum anderen der Zugriff auf einen Laufzeitdatensatz (dynamischer Kontext), der die lokalen Variablen und den Operandenstapel enthält. Dieser *Rahmen* (Frame) muss beim Aufruf einer Methode erzeugt und beim Verlassen verworfen werden. Bei einem geschachtelten Methodenaufruf existieren mehrere Rahmen (u. U. bei Rekursion von ein und derselben Methode) gleichzeitig, sie werden daher in einem Rahmenstapel gehalten. Der Rahmenstapel enthält also alle notwendigen Informationen, die zur Ausführung eines Programmfadens (Thread) nötig sind, er existiert daher als Teil der Verwaltungsstruktur der VM für Threads (siehe Kapitel 7) auf dem Heap. Genausogut wäre es möglich, die Rahmen einzeln auf dem Heap abzulegen, was allerdings zwei Nachteile hätte:

1. Der Speicherdurchsatz steigt. Bei jedem Methodenaufruf muss ein Heap-Block angefordert werden, das kostet Zeit. Die Freigabe eines solchen Blocks kann zwar explizit erfolgen, um die Speicherbereinigung zu entlasten, eine erhöhte Fragmentierung des Speichers findet dennoch statt, so dass dadurch keine Rechenzeit eingespart wird (siehe Abschnitt 5.2).

2. Bedingt durch die Spezifikation der Parameterübergabe beim Methodenaufruf ist bei Verwendung eines Rahmenstapels eine Optimierung möglich (siehe Bild 6.1), bei einzelnen Rahmen jedoch nicht: Die Parameter werden im Operandenstapel der aufrufenden Methode abgelegt und erscheinen in der aufgerufenen Methode in den lokalen Variablen. Eine Überlappung dieser beiden Speicherbereiche erspart ein Umkopieren der Daten (im Bild durch schräge Linien angedeutet). Das reduziert auch die Redundanz auf dem Heap.

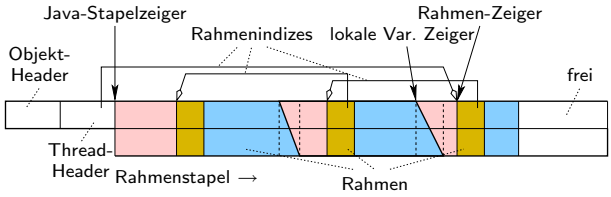


Bild 6.1: Der Rahmenstapel als Teil der Thread-Struktur

Natürlich hat die Verwendung eines Rahmenstapels auch einen Nachteil:

- Der Rahmenstapel hat eine vordefinierte Größe und der freie Bereich stellt einen Speicherplatzverschnitt dar. Um diesen Nachteil zu minimieren wird der Rahmenstapel auf der Thread-Struktur mit einer kleinen initialen Größe angelegt (ausreichend für vier bis fünf durchschnittlich umfangreiche Methodenaufrufe). Sie kann dann bei Bedarf vergrößert werden. Dazu wird die komplette Thread-Struktur, ergänzt durch den vergößerten Rahmenstapel, kopiert und anschließend der neue Thread in den alten Handle eingetragen; dieser Vorgang erscheint für die VM also transparent. Kurzfristig existieren beide Varianten des Threads auf dem Heap, was seinerseits zu einem Speicherengpass führen kann, umso mehr, je größer der Rahmenstapel ist.

Wie in Bild 6.1 gezeigt, besteht ein Rahmen aus drei Bereichen, den lokalen Variablen (von denen der untere Teil die Parameterliste enthält), dem eigentlichen Rahmen-Datensatz und dem Operandenstapel der Methode. Da Rahmen ihrerseits auf einem Stapel liegen, kann ein Rahmen als auf dem Operandenstapel des Rahmens der aufrufenden Methode angesehen werden. Die Verwaltung der Stapel ist daher zusammengefasst und es existiert nur ein Stapelzeiger je Thread, der in der Thread-Struktur gesichert wird. Rahmen werden durch einen Index im Stapel referenziert, der Thread sichert dabei den Index des Rahmens der gerade in Ausführung befindlichen Methode. Die Rahmen-Struktur sichert den eigenen `JPC` und den Index des Rahmens der aufrufenden Methode (falls vorhanden). Zusätzlich spiegelt der Rahmen statische Informationen der Methodendeklaration in der Klasse für einen schnellen Zugriff: Das ist z. B. die Adresse im internen oder externen Festwertspeicher des Code-Einsprungpunkts der Methode. Der Bytecode-Interpreter benötigt sie, da manchmal Referenzen auf Code-Blöcke relativ zum Methodenbeginn angegeben werden. Der Rahmen enthält auch die Anzahl der lokalen Variablen und Methoden-Flags, die zur Laufzeit der Methode oder beim Beenden benötigt werden (z. B. `synchronized`). Schließlich ist noch eine Referenz auf das Klassenobjekt, in der die Methode deklariert ist (der statische Anteil des Ausführungskontexts), enthalten. Diese Informationen könnten zwar auch durch Zugriff auf die Klassendatei ermittelt werden, sie sind dort jedoch in tiefen Strukturen vergraben und der Zugriff ist entsprechend aufwändig.[18]

[18] Der Zeiger auf den Code-Einsprungpunkt ist durch die CPR der Klasse der aufrufenden Methode zu ermitteln. Dazu muss der Index im Konstantenpool gesichert werden. Die Flags und die Parameterzahl können zu einem Byte gepackt werden, das zusammen mit der Referenz auf den Rahmen der aufrufenden Methode in ein Wort gelegt wird. Der Mehrverbrauch von Stapelspeicher beträgt also maximal ein Wort.

6.2.1 Selektierung

Beim Betreten eines Rahmens wird dieser selektiert, d. h. die Information in der Rahmen-Struktur wird ausgelesen und in VM-internen Variablen gesichert. Dabei werden Indizes der Strukturen in Zeiger aufgelöst, so dass ein direkter Zugriff möglich ist. Es existiert also je ein Zeiger auf den Rahmenstapel des Threads, den aktuellen Rahmen, die lokalen Variablen des aktuellen Rahmens und den als nächstes auszuführenden Bytecode (`JPC`). Die Klasse des aktuellen Rahmens kann auf Anfrage ebenfalls selektiert werden, dies resultiert in einen Zeiger auf dessen CPR. Die virtuelle Maschine verwendet und manipuliert nun ausschließlich die Daten der Selektion. Beim Verlassen eines Rahmens, sei es durch Aufruf einer Methode (Push) oder durch einen Rücksprung (Return/Pop), wird die Selektion entsprechend in den Rahmen gesichert und der Rahmen des Aufrufers/Aufgerufenen selektiert. Kehrt der Bytecode-Interpreter zum Aufrufer zurück, so werden zusätzlich die Selektionsdaten in den Thread gesichert.

Das Konzept der Selektierung, also die kurzfristige Auslagerung von Datensätzen in einfach zugreifbare VM-interne Variablen allgemein, wird auch bei anderen Strukturen angewendet. Insbesondere zu nennen sind neben Rahmen und Threads auch Objekte und Klassen.

6.3 Das Native-Interface

Java definiert keine Möglichkeit, mit der Hardware der implementierenden Plattform zu interagieren. Dies ist gerade bei eingebetteten Systemen, die für Steuerungsaufgaben vorgesehen sind, aber unbedingte Voraussetzung. Bereits in Abschnitt 6.1 wurde beschrieben, wie *nativer Code* (z. B. in Interrupt-Routinen) der virtuellen Maschine Signale senden kann. Für den von Java ausgelösten Aufruf nativen Codes werden in der Spezifikation [LY00] native Methoden (kurz) beschrieben. Dieser Ansatz bietet die Möglichkeit, nativen Code statt einer gewöhnlichen Java-Methode ausführen zu lassen. Dafür existiert der Methodenmodifier *`native`*, der den Java-Compiler anweist, Methodendeklarationen ohne eingebundenen Bytecode zu erzeugen. Wie der native Code an diese Methoden gebunden wird, ist nicht vorgeschrieben. Praktisch hat dies unter dem Gesichtspunkt der kleinen eingebetteten Systeme einige Nachteile:

- Neben den Klassendateien mit ihren statischen und Laufzeit-Strukturen muss eine weitere Hierarchie für die nativen Methoden verwaltet werden.

- Die Bytecodes für den Methodenaufruf müssen das `native`-Flag der Methoden auswerten, eine Fallunterscheidung treffen und die zweite Hierarchie geeignet nach dem nativen Implementierer durchsuchen (bzw. eine geeignete Tabellenstruktur verwenden).
- Die Granularität der Verwendung nativen Codes ist auf die Ebene der Methoden beschränkt. In Abschnitt 4.2.1 wurde beschrieben, welchen Vorteil Java-Bytecode im Gegensatz zu nativem Code im Bezug auf die benötigte Code-Größe für einen Algorithmus hat. Dieser Vorteil muss innerhalb von nativen Methoden aufgegeben werden.
- Da in dieser Implementierung kein unterliegendes Betriebssystem das Thread-Scheduling übernimmt, sondern die virtuelle Maschine selbst dies auf Bytecode-Ebene tut, versagt dieser Mechanismus bei nativen Methoden. Diese sind dann von der virtuellen Maschine nicht unterbrechbar und native Methoden, die viel Rechenzeit benötigen (insbesondere bei Schleifen), zerstören die gleichmäßige und verhersagbare Bearbeitung mehrerer Threads. Dies ist allerdings Voraussetzung für eine Echtzeitunterstützung.
- Der Aufruf von Java-Methoden vom nativen Code gestaltet sich aufwändig. Es muss ein Mechanismus geschaffen werden, den Bytecode-Interpreter in einem eigenen Ausführungskontext neu zu starten. Die Kontrolle darf dann nicht an den Scheduler abgegeben werden, da dann der Kontext der nativen Methode mit unvorhersagbaren Effekten verlassen würde.

6.3.1 Native Code-Blöcke im Java-Bytecode

Diese Nachteile vermeidet ein Native-Interface, das statt auf Methodenebene auf Bytecodeebene funktioniert. Statt den Methodenaufruf zu verwenden, um in nativen Code zu verzweigen, wird ein spezieller Bytecode verwendet. Für solche implementierungsabhängigen Fälle wurden in der Spezifikation [LY00] Bytecodes reserviert, so dass es bei zukünftigen Erweiterungen des Bytecode-Satzes nicht zu Konflikten kommen kann. Die Implementierung und Ausführung nativer Methoden unterscheidet sich also nicht von der gewöhnlicher bytecodebasierter Methoden, bis ein *callnative*-Bytecode auftritt. Dieser springt den direkt dem Bytecode folgenden nativen Code an, dazu muss nur der aktuelle Wert des `JPC` ausgelesen und in den nativen Programmzähler eingetragen werden. Analog dazu existiert eine native Routine *leavenative*, die – in den nativen Code eingebunden – beim Erreichen den `JPC` auf die folgende Speicherzelle setzt und zum Bytecode-Interpreter zurückkehrt. Mithilfe dieser beiden Anweisungen kann beliebig zwischen den beiden Code-Welten gewechselt

werden. Der nachfolgende Quelltext-Ausschnitt demonstriert dieses Vorgehen (Java-Bytecodes und die `leavenative`-Codesequenz sind als Makros definiert, um die Lesbarkeit des Quelltextes zu erhöhen; ein vollständiges Code-Beispiel findet sich in Anhang B.1):

```
...
aload_0                     ; Bytecodes...
invokevirtual  <init>()V
CallNative                  ; auf nativen Code schalten
ld             a,#42        ; nativer Code...
PushOps        ,a
LeaveNative                 ; auf Bytecode schalten
ireturn                     ; Bytecode...
```

Es können also die Vorteile beider Kodierungen ideal kombiniert werden und je nach Wunsch immer die Variante mit dem kompakteren oder dem schnelleren Code verwendet werden. Aus Sicht des Bytecode-Interpreters ist jeder native Block mit einem Bytecode zu vergleichen, nur dass bei jeder Implementierung ein anderer Inhalt steht. Es ergibt sich dadurch eine feinere *Code-Granularität*.

Der Datenaustausch erfolgt ähnlich einfach: Dem nativen Code stehen alle Datensätze zur Verfügung, die auch der Bytecode-Interpreter kennt, also auch der gerade selektierte Rahmen (siehe Abschnitt 6.2.1). Mittels kleiner Makros kann auf den Operandenstack (im Beispiel oben zu sehen) und die lokalen Variablen zugegriffen werden. Da der Zugriff auf Klassen- und Objektvariablen relativ wenig Aufwand bedeutet, kann dieser auch direkt im nativen Code erfolgen, dazu muss zunächst lediglich die entsprechde Klasse oder das entspechende Objekt selektiert werden, um für mehrere Zugriffe die entsprechenden Zeiger zu erhalten. Komplexere Zugriffe auf Java-Datenstrukturen sind vom nativen Code aus weder sinnvoll (zu aufwändig) noch nötig. Einfacher ist es, jeweils kurz auf die Java-Seite zu wechseln und einige Bytecodes auszuführen, bestes Beispiel dafür ist der Aufruf von Java-Methoden.

Etwas schwieriger gestaltet sich dieses Zusammenspiel bei der Entwicklung. Wünschenswert ist auch hier auf Quelltextebene eine enge Koppelung zu ermöglichen und es dem Entwickler zu erleichtern, nativen Code mit der gewünschten feinen Granularität einzubinden. Vorhandene Java-Compiler kennen dieses Zielsystem nicht und schon gar nicht den `callnative`-Bytecode und sie können auch keinen nativen Code direkt einbinden. Eine Hintereinanderschaltung von Java- und nativen Compiler löst dieses Problem zumindest teilweise. Der Vorverlinker (siehe Abschnitt 4.2.5) kann als Zielsystem auch Quelltext erzeugen, was durch ein entsprechendes Geräteprofil gelöst ist. Der erzeugte Quelltext enthält die Archiv-Datenstruktur mit den Java-Klassen in Form von Datensätzen. Wird dieser mit dem nativen Compiler des Zielsystems übersetzt, so

entsteht das gewünschte Binär-Abbild des Archivs, das die JavaVM verarbeiten kann und die interne Speicherbank bildet. Der Code für die nativen Methoden wird dabei aus speziell formatierten Quelltexten, die den Klassennamen entsprechend benannt sind, ausgelesen und in den Ziel-Quelltext integriert (*Inlining*). In diesen Dateien können die nativen Methoden unter Verwendung der Programmiersprache des Zielsystems implementiert werden. Das Format der dazu definierten Platzhalter und Hilfsstrukturen der Referenzimplementierung wird in [Böh02] zusammengestellt.

Dieses Verfahren ist nicht optimal. Es müssen bei der Entwicklung immer zwei Quelltext-Sätze synchron gehalten und die Fehlermeldungen von drei Werkzeugen (Java-Compiler, Vorverlinker, Assembler/Compiler des Zielsystems) ausgewertet und zugeordnet werden. Die Einbindung von Java-Code (Quelltext) in native Methoden ist bei diesem Verfahren in der Regel auch nicht möglich, es müssen die erforderlichen Java-Bytecodes als Datensätze in den nativen Quelltext eingebunden werden. Hierbei sind Erweiterungen der Entwicklungswerkzeuge denkbar, die es ermöglichen, Basic-Blocks von Java-Quellcode in nativen Methoden zu verwenden. Diese Blöcke werden dann durch separate Aufrufe des Java-Compilers einzeln in Bytecodes übersetzt. Sprünge aus einem solchen Block heraus sind allerdings nicht definiert.[19]

6.4 Behandlung von Ausnahmen

Zur Behandlung von Fehlerzuständen in Programmen werden in Java Ausnahmen (*Exception*s) verwendet. Dabei ist ein Fehlerzustand mit einem speziellen Objekt verknüpft, dessen Typ darüber entscheidet, welche Code-Sequenz zur Fehlerbehandlung (`catch`-Block) einer überwachten Code-Sequenz (`try`-Block) herangezogen wird. Ist keine passende Fehlerbehandlung vorhanden, so wird der Fehlerzustand zur aufrufenden Methode weitergereicht und ggf. dort bearbeitet. Auf der Ebene der virtuellen Maschine wird im Fall eines Fehlerzustands eine Tabelle abgearbeitet, die den definierten Fehlerzustands-Typen (Überprüfung ähnlich wie beim `instanceof`-Bytecode) und überwachten Code-Bereichen entsprechende Einsprungpunkte im Bytecode der bearbeitenden Methode zuordnet. Dieser Mechanismus lässt sich mit relativ geringem Aufwand auch bei kleinen Zielsystemen umsetzen.

Problematisch ist dieses Verfahren mit steigender Anzahl von Fehlerzustands-Typen, da für jeden eine entsprechende (wenn auch kleine) Klasse auf dem

[19] Die neue Version 5.0 des Java Development Kits ermöglicht nun die Einbindung von Meta-Informationen in den Java-Quelltext (die auch in die Klassendateien integriert werden), Annotations. Dieser Ansatz wird in Zukunft die Integration von nativen Code direkt in einen Java-Quelltext ermöglichen. Das soll in Kapitel 10 näher betrachtet werden.

Zielsystem abgelegt werden muss. Die Größe des Zielsystems begrenzt also die Möglichkeiten der Fehlerbehandlung. Ferner muss, um eine Ausnahme auszulösen, ein entsprechendes Objekt erzeugt werden, dessen Typ als symbolische Referenz im Konstantenpool einer Klasse definiert ist. Dies macht die Ausnahmenerzeugung in nativen Methoden umständlich (wichtig, um den Fehlerzustand beim Zugriff auf Peripheriekomponenten anzuzeigen) und im Kern der virtuellen Maschine beinahe unmöglich. Das Problem ist die Objekterzeugung, die an eine bestimmte Klasse gekoppelt ist, die zunächst auf der Ebene des Java-Bytecode initialisiert werden muss (siehe Abschnitt 8.1.2). Da der Kern direkt keine Klasse initialisieren kann, können keine Ausnahmen von VM-internen Fehlerzuständen erzeugt und der ablaufenden Anwendung zugeordnet werden. VM-interne Ausnahmen (z. B. `OutOfMemoryError`) werden daher mit Fehlercodes behandelt, die nach einem Neustart der VM in einem `ErrorHandler` bearbeitet werden (diese Vorgehensweise wird noch in Abschnitt 8.1 beschrieben). Anhang C.4 listet die zielsystemabhängigen Fehlercodes auf der Referenzimplementierung auf.

6.5 Leistungsfähigkeit

Eine interpreterbasierte virtuelle Maschine auf einem 8-Bit-Mikrocontroller ist bei rechenintensiven Anwendungen naturgemäß überfordert. Um die Arbeitsgeschwindigkeit eines solchen Systems abzuschätzen, sollen einige Messungen durchgeführt werden. Tests am eigenen eingebetteten System mit etablierten Benchmark-Suites (z. B. CaffeineMark [22], SpecJVM98 [23]) sind schwierig. Entweder sind diese Frameworks nicht im Quelltext verfügbar, also nicht portierbar und nur auf der J2SE lauffähig oder die Tests sind zu umfangreich, zu rechenintensiv oder setzen Funktionalitäten voraus, die das kleine System nicht bieten kann. Daher wurde für dieses System ein eigener Satz von Benchmark-Programmen entworfen.

Grundsätzlich können folgende Testverfahren unterschieden werden:

Mikrobenchmarks Kernkomponenten der VM oder des gesamten Laufzeitsystems (Bytecodes, Threads, Speicherverwaltung) werden untersucht.

Algorithmische Benchmarks Typische Aufgaben aus der Informatik (algorithmische Berechnungen, Listen- und Baumverwaltung, Sortierung, Bearbeitung von Zeichenketten, Datenkompression) werden bearbeitet.

Anwendungsorientierte Benchmarks Bestimmte Anwendungsfälle (Netzwerk-, Datenbankzugriff, grafische Ausgaben) oder komplette Anwendungen mit bestimmten Testdaten werden abgearbeitet.

Für eine Einordnung der Leistungsfähigkeit des Interpreters wurden einige Mikrobenchmarks (siehe Anhang E.4) implementiert. Bei jedem Test werden 10000 Schleifendurchläufe mit nur kurzen Test-Sequenzen abgearbeitet. Erfasst wurden dabei die Ausführungszeiten für spezifische Bytecode-Klassen. Von den Abarbeitungszeiten wurden anschließend die von entsprechend vielen Durchläufen mit leerem Schleifenrumpf abgezogen, die Schleifenbearbeitung selbst geht also nicht in die Performanceanalyse ein. Diese Tests wurden das erste Mal bereits in [Böh98] durchgeführt, Tabelle 6.1 zeigt hingegen aktualisierte Werte.

Tabelle 6.1: Mikrobenchmarks für Bytecode-Klassen, ausgeführt auf dem Yogi2-Referenzsystem

Test	**Zeit** [ms]		**Anzahl**		**Durchsatz**		**Zyklen**	
	Real	User	BCs	OPs	BCs/s	OPs/s	cy/BC	cy/OP
BenchVMbase*	2212	2157						
BenchVMarithm	4000	3997	14	6	35026	15011	228	533
BenchVMcond	1203	1196	4	1	33445	8361	239	957
BenchVMarray	1626	1616	4	1	24752	6188	323	1293
BenchVMsvar	2843	2835	4	2	14109	7055	567	1134
BenchVMivar	3499	3488	6	2	17202	5734	465	1395
BenchVMsmet†	3368	3356	3	1	8939	2980	895	2685
BenchVMimet‡	4132	4122	3	1	7278	2426	1099	3298
BenchVMobj	15379	13473	6	3	4453	2227	1796	3593
BenchVMobj§	6771	5229	1	1	1912	1912	4183	4183
`NOP`¶					170213		47	

*Leere Schleife, Grundlage aller weiteren Messungen
†Ein Parameter
‡Ohne zusätzlichen Parameter (nur `this`)
§Bereinigt von zwei Methodenaufrufen (Konstruktor), entspr. 2×BenchVMimet
¶Aus dem Quelltext: 5+3+6+6+10+6+6+2+3 Taktzyklen (8 MHz), siehe Anhang B.2

Zusätzlich zu den Realzeitmessungen ist es mittels des in Abschnitt 5.3 eingeführten Verfahrens möglich, nur die vom Anwendungsthread (User) verwendete Rechenzeit zu erfassen und von der anderer Aufgaben (z. B. Speicherbereinigung) zu trennen. Ferner wurden die Benchmarks entsprechend der verwendeten Bytecodes bewertet und in für den Programmierer erkennbare Einheiten unterteilt. Es wurden also mehrere Bytecodes, die immer als Einheit zu betrachten sind, zu einer Operation zusammengefasst und entsprechende Vergleichswerte ermittelt. Die Werte der Anwendungszeitmessungen wurden schließlich in Ausführungsgeschwindigkeiten umgerechnet (Bytecode- bzw. Operationen-Durchsatz). Die letzten beiden Spalten zeigen die aus der Taktfrequenz errechneten durchschnittlichen Taktzyklen des Zielsystems, die zur Ausführung eines Bytecodes oder einer Operation nötig sind. Zusätzlich zu diesen Messun-

gen wurde noch die obere Schranke der Ausführungsgeschwindigkeit auf dem Zielsystem bestimmt. Diese ist begründet durch den Schleifen- und Interpreteroverhead, der je ausgeführten Bytecode immer auftritt. Dazu wurden die benötigten Zyklen der verwendeten Maschinenbefehle zur Ausführung des `NOP`-Bytecodes auf dem Zielsystem bestimmt und zusammengezählt.

Die Messungen von Real- und Anwendungszeit unterscheiden sich nur geringfügig und lassen einen geringen Overhead bei der Rechenzeitzuteilung auf die Java-Threads (siehe nächstes Kapitel) erkennen. Ein nennenswerter Unterschied ist lediglich bei der Objekterzeugung erkennbar, da dies die einzige Operation ist, die sich auf den Java-Heap auswirkt und hierbei auch die Speicherbereinigung ins Gewicht fällt. Aus dem Bytecode-Durchsatz lässt sich die Leistungsfähigkeit des Systems mit einem Java-Prozessor bei einer Taktfrequenz von 20 – 100 kHz gleichsetzen (bei dort angenommenen durchschnittlich zwei Taktzyklen je Bytecode [Sch05, Sch06]). Die Anzahl der nötigen Zyklen je Operation spiegelt die Komplexität in der jeweiligen Klasse wider. Einfache Operationen auf dem Stack werden am schnellsten ausgeführt, Zugriffe auf Objekte sind geringfügig aufwändiger. Ein Methodenaufruf (inklusive -rücksprung) hat bereits umfangreichere Manipulationen am Stack und am Thread zur Folge (ein Rahmen wird erzeugt und verworfen). Die Objekt-Erzeugung mit dem `new`-Operator wird in diesen Messungen nur geringfügig aufwändiger eingestuft, als ein Methodenaufruf.

Schon am Interpreteroverhead ist zu erkennen, dass der Rechenkern des gewählten Zielsystems für die Interpretierung von Bytecode nur bedingt geeignet ist. Für zukünftige Implementierungen leistungsfähigerer Systeme ist eine Architektur mit weniger Zyklen je Maschinenbefehl und mit Befehlen, die besser an die 16- oder 32-Bit-Wortbreite von Java angepasst sind, vorzuziehen. Vergleichsmessungen mit Java-Implementierungen u. a. auf Arbeitsplatzsystemen verdeutlichen dies noch einmal (siehe Bild 6.2). Neben dem hier vorgestellten Referenzsystem der Yogi2-VM auf dem ST7 wurden die Tests auch auf dem Coldfire-Prozessor mit der verwandten JCVM32 (siehe Abschnitt 9.2) und auf moderner PC-Hardware mit den von *Sun* verfügbaren virtuellen Maschinen, die im interpretierenden Modus gestartet wurden, durchgeführt. Natürlich handelt es sich dabei um eine ganz andere Klasse sowohl von Prozessoren, als auch von virtuellen Maschinen, so dass dieser Vergleich nur für einen Trend herangezogen werden sollte. Um aussagekräftige Vergleichswerte zu erhalten, wurde (mit Außnahme des ST7) die reale Ausführungszeit ermittelt und auf die nominellen Taktfrequenzen der Zielsysteme normiert; dies resultiert jeweils in einen Wert, der mit den benötigten Taktzyklen je Java-Operation vergleichbar ist.

Es zeigt sich, dass zusätzlich zum besseren Prozessor-Instruktionssatz Optimierungen wie Caching und Superskalarität sich enorm auf die Ausführungs-

geschwindigkeit auswirken. Ein Vergleich mit anderen virtuellen Maschinen im Bereich eingebetteter Systeme und im speziellen bei 8-Bit-Implementierungen wäre hier wünschenswerter, scheitert aber an deren Verfügbarkeit oder der Verfügbarkeit geeigneter Zielsysteme.

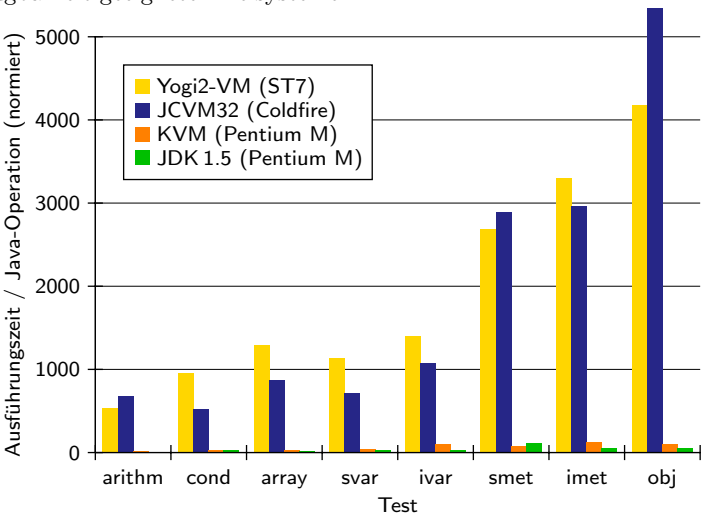


Bild 6.2: Interpreter-Leistungsfähigkeit im Vergleich

Kapitel 7

Multithreading

Bei eingebetteten Systemen, auf denen nur die JavaVM läuft, haben *Thread*s (nebenläufige Programmfäden) eine besondere Bedeutung. Sie stellen die Prozesse eines Betriebssystems dar und werden nicht nur auf Anwendungsebene genutzt, sondern auch zum Betrieb des Systems. Sowohl Komponenten der VM (Speicherbereinigung, Klasseninitialisierung) als auch der Laufzeitbibliothek (z. B. die Verwaltung von Puffern und Event-Queues von Hardware-Schnittstellen) verwenden Threads. Ferner gibt es bei eingebetteten Systemen erweiterte Anforderungen an die Threads und an die Einheit, die für die Rechenzeitzuteilung zuständig ist, den *Scheduler*. Oft ist eine Echtzeitverarbeitung von Ereignissen, d. h. mit definiertem Zeitverhalten, gefordert. Bild 7.1 zeigt das gegenüber [Tan01] erweiterte Zustandsdiagramm der in der Yogi2-VM implementierten Threads.

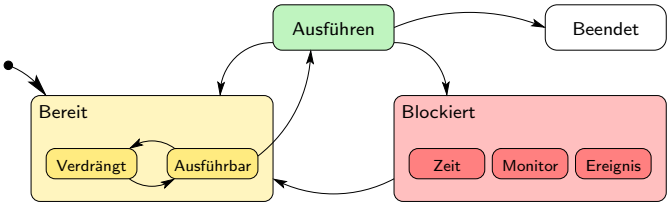


Bild 7.1: Die Zustände eines Threads

Die parallele Ausführung mehrerer Threads ist in die Sprache Java fest integriert. Die Koppelung der Programmiersprache an die virtuelle Maschine ist aber nur lose definiert. Es gibt nur ein Schlüsselwort in der Sprache und zwei explizite Bytecodes zur Synchronisierung und der Rest der Thread-Konfiguration findet über das Native-Interface bzw. über die Laufzeitbibliothek statt. Code zur Thread-Synchronisierung ist direkt in `java.lang.Object` eingebaut, steht also allen Klassen zur Verfügung. Zum Erzeugen, Starten und Verwalten eigener Threads dient die Klasse `java.lang.Thread`. Die Schnittstelle der Threads

zur virtuellen Maschine ist in jedem Fall eine Instanz von `Thread`, auf die durch die genannten Mechanismen zugegriffen wird (der Aufbau der Thread-Struktur wurde bereits in Abschnitt 6.2 skizziert).

7.1 Threads auf Interpreterebene

Bereits im Abschnitt 6.1 wurde erwähnt, dass hier Threads auf Interpreterebene verwaltet werden. Ein Thread-Wechsel findet also immer zwischen Bytecodegrenzen statt. Dieses Konzept der atomaren Bytecodes vereinfacht nicht nur die Verwaltung der Datenkonsistenz bei Thread-Wechseln (so sind keine lokalen Kopien der Daten je Thread nötig, die mit dem globalen Datenbestand synchronisiert werden müssen) auch die Thread-Wechsel selbst können sehr einfach und portabel realisiert werden. Es werden dabei keine Thread- oder Prozess-Wechselmechanismen eines unterliegenden Betriebssystems benötigt und auch keine dafür ausgelegten Hardwarekomponenten eines Prozessors verwendet. Für die Abfertigung der Threads wird lediglich eine konstante Zeitbasis benötigt, die in der Lage ist, mit festgelegter Periode Ereignisse auszulösen. Die meisten Mikrocontroller haben dafür geeignete Hardware (Timer).

7.1.1 Abfertigung der Threads

Die *Abfertigung* der Programmfäden (Thread-Dispatching) ist fest in den Kern der VM integriert. Der Thread-Dispatcher stellt die Hauptschleife der Java-VM dar, nur hier wird zu den einzelnen Threads verzweigt. Für die Auslösung von zeitgesteuerten Ereignissen wird ein Timer des Mikrocontrollers verwendet, der auf einen Millisekundentakt programmiert wird. Tritt ein Timer-Event auf, werden Bedingungen (Zähler, Hardware-Zustände) geprüft und ggf. entsprechende Signale (Bytecode-Interrupts, siehe Abschnitt 6.1) an die VM übergeben. Die Millisekunde als Zeitbasis hat sich als praktikabel herausgestellt, ist aber natürlich implementierungsabhängig. Als Abbruchkriterium für die Bytecode-Interpretierung eines Threads ist das Schedule-Flag definiert worden, welches in diesem Zeitraster frei programmiert werden kann. Im Normalfall wird dafür eine feste *Zeitscheibe* bestimmt, die als ein guter Kompromiss zwischen einer scheinbaren Gleichzeitigkeit der Thread-Ausführung (möglichst klein) und der Minderung des Scheduling-Overheads (möglichst groß) gewählt wird [Tan01][20]. Um die Reaktivität zu erhöhen, legt diese Zeitscheibe lediglich ein Maximum fest, Ereignisse in der Gegenwart oder in der Zukunft können das

[20] In dieser konkreten Implementierung wurde eine Zeitscheibe von 64 ms festgelegt. Messungen hierzu finden sich in Abschnitt 7.4.

Abbruchkriterium auf einen früheren Zeitpunkt setzen. Der konkrete Ablauf bei der Ausführung mehrerer Threads ist im Folgenden noch einmal schrittweise dargestellt:

1. Start des Thread-Dispatchers.
2. Der Scheduler wird aufgerufen, dieser untersucht die Liste der Threads auf dem Heap und wählt entsprechend des festgelegten Scheduling-Algorithmus (siehe Abschnitt 7.1.3) einen geeigneten Thread aus.
3. Gibt es keine aktiven Threads mehr (Threads im Zustand *Beendet* werden von der Speicherbereinigung entfernt), wird die Hauptschleife verlassen und die VM neu gestartet. Der Neustart ist bei eingebetteten Systemen, bei denen die JavaVM die einzige Anwendung ist, sinnvoll.
4. Warten alle Threads (sind im Zustand *Blockiert*), so wird der Zeitpunkt ermittelt, an dem der nächste wartende Thread wieder rechnen möchte, und der Controller in den Wartezustand versetzt (dabei können auch vorhandene Stromspar-Mechanismen angewendet werden). Dieser Zeitpunkt kann auch später liegen, als von der maximalen Zeitscheibe bestimmt. Wurde die Warteperiode beendet, so wird der Vorgang wieder bei 2 fortgesetzt, ggf. werden zunächst gesetzte Bytecode-Interrupts bearbeitet.
5. Möchte ein Thread rechnen (ist im Zustand *Bereit* und außerdem vom Scheduler ausgewählt worden), so wird der nächstgelegene Zeitpunkt ermittelt, an dem einer der übrigen wartenden Threads geweckt werden soll (sofern bekannt). Liegt er vor dem durch die Zeitscheibe bestimmten Zeitpunkt, wird er stattdessen verwendet, um den Bytecode-Interpreter zu unterbrechen.
6. Der Thread wird selektiert und der Bytecode-Interpreter gestartet. Er kehrt dann selbstständig zum vorgesehenen Zeitpunkt oder durch ein anderes Ereignis gesteuert zurück.
7. Die Bedingungen der Rückkehr werden geprüft (Bytecode-Interrupt-Flags), ggf. werden die entsprechenden Bytecode-Interrupts bearbeitet und anschließend noch einmal das Schedule-Flag geprüft.
8. War ein Abbruch des aktuellen Threads gefordert, so werden die Selektionsdaten in das Thread-Objekt gesichert und der Scheduler wird wieder bei 2 gestartet. Andernfalls wird dieser Thread bei 6 fortgesetzt.

Die auf dem Heap liegenden Thread-Objekte sind zu einer Liste verkettet, der letzte Thread weist dabei wieder auf den ersten. Bei jedem Durchlauf des Schedulers wird diese Liste maximal zwei Mal durchsucht. Einmal, um den Zeitpunkt für das Abbruch-Ereignis festzustellen und Thread-Daten für

den Scheduler zu sammeln und ggf. ein zweites Mal, um den auszuführenden Thread anhand dieser Daten durch den Scheduler auszuwählen (das wird in Abschnitt 7.3 noch ausführlich behandelt). Dieses mehrmalige Durchsuchen der Liste stellt einen Engpass bei der Abfertigung dar. Die Verwendung von mehreren sortierten Listen für Threads der Zustände *Blockiert*, *Verdrängt* und *Ausführbar* kann die Auswahl eines geeigneten Threads beschleunigen. Dabei wird allerdings die rechenzeitintensive Arbeit lediglich verlagert. Die bearbeiteten Threads müssen entsprechend wieder in die Listen einsortiert werden. Die Listenverwaltung wird komplizierter und es wird mehr Code benötigt.

Sollen hier Optimierungen zum Tragen kommen, z. B. eine binäre Suche beim Einsortieren in die Listen verwendet werden, so sind weitere Verwaltungsstrukturen mit Referenzen auf die Threads auf dem Heap nötig, die zudem bei fester Größe die Anzahl der Threads begrenzen. Auch wird es schwierig, wenn sich die Zustände bereits in die Listen einsortierter Threads ändern, diese müssten dann einer Liste entnommen werden und dann an anderer Stelle einsortiert werden. Die Verwendung einer linearen Liste aller Threads hat diese Probleme nicht und spart durch die einfache Implementierbarkeit wertvollen nichtflüchtigen und flüchtigen Speicherplatz. Eine Verwendung von mehreren Thread-Listen bleibt größeren Systemen vorbehalten, die viele Threads ausführen. Bei kleinen Systemen bleibt die Anzahl zu bearbeitender Threads schon durch den Speicherplatz begrenzt und hier mögliche Optimierungen greifen nicht.

7.1.2 Synchronisierung

Ein weiterer wichtiger Punkt ist eine schlanke Realisierung der Thread-*Synchronisierung*, welche in der Programmiersprache Java über das `synchronized`-Schlüsselwort gesteuert wird. Es kann einer Methode oder einem Code-Block ein Objekt zugeordnet werden, das in einem *Monitor* eine Zugangsberechtigung für die Threads verwaltet (gegenseitiger Ausschluss). So ein Monitor besteht aus einer Referenz auf den Thread, der den Monitor erworben hat, und einem Zähler für die Anzahl der Erwerbungen (bei zyklischem Aufruf). Diese Information ist im Header jeder auf dem Heap residenten Objektinstanz vorhanden und schlägt beispielsweise auf der Referenzimplementierung mit drei Bytes je Objekt zu Buche. Das liegt in der Größenordnung des in [BKMS98] vorgeschlagenen Schemas, allerdings werden dort in einigen Fällen externe Monitor-Objekte ergänzt. Hier werden Vereinfachungen vorgenommen, so dass dies nicht nötig wird.

Kommt es zu einem Konflikt, also wenn ein Thread einen Monitor erwerben möchte, der bereits einem anderen Thread zugeordnet ist, so wird der nicht berechtigte Thread in den Zustand *Blockiert/Monitor* versetzt und bleibt es, bis

der blockierende Thread den Monitor freigegeben hat. Die Zugangsberechtigung wird dabei immer beim Aufruf einer synchronisierten Methode oder beim Auftreten des `monitorenter`-Bytecodes geprüft. Der blockierende Thread kann den Monitor eines Objekts auch zeitweise freigeben, wenn die Methode `wait()` dieses Objekts aufgerufen wird. Die Sprache Java sieht nun vor, einen dieser für eine unbestimmte Zeit wartenden Threads mit `notify()` explizit aufzuwecken, wofür eine Warteschlange vorhanden sein muss. Bei dieser Implementierung wird dieser Vorgang derart vereinfacht, dass auf Warteschlangen verzichtet und nur `notifyAll()` unterstützt wird.

Wird eine Blockade durch einen Monitor gelöst, so werden in dieser Implementierung *alle* Threads benachrichtigt, die im Zustand *Blockiert/Monitor* sind. Sie werden dann anschließend, wenn sie vom Thread-Dispatcher aufgerufen werden, die Bedingung erneut prüfen, die sie blockieren ließen. Sie werden erneut den synchronisierten Methodenaufruf versuchen, den `synchronized`-Bytecode aufrufen oder beim Verlassen von `wait()` versuchen, den alten Monitorzustand herzustellen. Diese Wiederholung der Bedingungsprüfung der Reihe nach, um dann ggf. wieder zu blockieren (*Spin-Locking*), hat sich als sehr effektiv und vor allem einfach und speicherplatzeffizient in der Realisierung herausgestellt.

7.1.3 Faires Scheduling

Der Sprachstandard Java bzw. die Spezifikation für die virtuelle Maschine [LY00] sieht nur einen prioritätsbasierten Scheduler vor. Definiert wurden lediglich die Prioritäten 1 bis 10, um die Rechenzeit auf Threads im Zustand *Bereit* zu verteilen (zur besseren Unterscheidung werden diese Werte im Folgenden als *Java-Priorität*en bezeichnet). Dabei werden keine Vorschriften gemacht, wie sich diese Prioritäten genau auswirken sollen. Es kommen bei Java-Systemen grundsätzlich zwei Möglichkeiten in Betracht [Tan01]:

Priority Scheduling Ein Thread mit einer höheren Priorität wird grundsätzlich vor allen anderen Threads mit einer niedrigeren Priorität ausgeführt. Damit diese laufen können, müssen alle Threads mit einer höheren Priorität blockieren oder die Rechenzeit freiwillig abgeben (`yield()`).

Fair-Share Scheduling Die Priorität eines Threads gibt eine Richtlinie für die Wahrscheinlichkeit der Ausführung an. Ein Thread mit einer niedrigeren Priorität wird dann um einen festgelegten Faktor langsamer laufen, als ein Thread höherer Priorität. Fair-Share Scheduling kann als Priority-Scheduling mit dynamischen Prioritäten angesehen werden.

Hauptvorteil beim Priority Scheduling ist die Vorhersagbarkeit der Thread-Wechsel aus Programmierersicht, da ein Wechsel i. d. R. nur vom gerade ausgeführten Thread angestoßen werden kann. Hauptvorteil beim Fair-Share Scheduling ist die höhere Gerechtigkeit bei der Zuteilung der Rechenzeiten: Ein höher priorisierter Thread, der keine `yield()`s enthält, kann die anderen nicht komplett verdrängen.

Die Wahl fiel für die Yogi2-VM aus noch einem entscheidenden Grund auf Fair-Share Scheduling: der Wunsch einer nebenläufigen Speicherbereinigung, die auch mit einer bestimmten Regelmäßigkeit aufgerufen werden soll, wenn eine Anwendung die komplette Rechenzeit beansprucht. Beim Priority Scheduling müsste sie immer mit der höchsten Priorität laufen und selbst entsprechend die Rechenzeit abgeben.[21] Das Hauptproblem dabei ist, dass die Speicherbereinigung grundsätzlich in der Lage ist, jeden noch so dringlichen Thread zu blockieren auch dann, wenn dieser gar nicht auf eine Speicherfreigabe angewiesen ist. Beim Fair-Share Scheduling muss die Speicherbereinigung nur mit einer niedrigen Priorität ausgeführt werden und der Scheduler kümmert sich um die Zuteilung der Rechenzeit. Voraussetzung dafür ist ein zu jedem von der Speicherbereinigung erreichbaren Zeitpunkt konsistenter Heap (das wurde bereits in Abschnitt 5.2 erläutert).

Natürlich verliert der Programmierer beim Fair-Share Scheduling einen Teil der Kontrolle über die Verteilung der Rechenzeiten auf seine Threads. Die Vorhersagbarkeit der Ausführungszeiten ist aber gerade bei Anwendungen für eingebettete Systeme oft eine zwingende Voraussetzung. Es müssen also zusätzliche Maßnahmen geschaffen werden, dem Programmierer die Kontrolle über die Threads zu geben, die über die normalen Prioritäten hinausgehen, eine Echtzeiterweiterung von Java-Threads.

7.2 Echtzeiterweiterung (EDF-Scheduling)

Bereits im Jahre 2000 hat die Real-Time-Experts-Group die *Real-Time Specification for Java* (RTSJ) veröffentlicht [24]. Sie ist z. Zt. Teil des Java Community Process und kann in zukünftigen Versionen von Java Teil des Sprachstandards werden oder zumindest eine Erweiterung. Die RTSJ hat zum Ziel, Echtzeit-Fähigkeiten als eingenständiges Modul zur Verfügung zu stellen, das in die J2SE oder J2ME integriert werden kann. Größte Sorgfalt dabei wurde auf Kompatibilität, Portabilität und Erweiterbarkeit gelegt. Herausgekommen

[21] Tatsächlich wird das bei den meisten Java-Implementierungen so gemacht. Der GC blockiert für kurze Zeitspannen sämtliche Anwendungs-Threads an vordefinierten Zeitpunkten, an denen sie sich in einem konsistenten Zustand befinden (GC-Punkte) und bereinigt im besten Fall inkrementell den Speicher.

ist dabei ein recht umfangreiches Rahmenwerk, das allein durch seine Größe nicht für allzu kleine Systeme geeignet ist. Bei der Yogi2-VM war das Ziel eine wesentlich schlankere Echtzeiterweiterung, die wie die RTSJ ohne syntaktische Erweiterungen von Java auskommt. Es folgt die Beschreibung einiger Kernkomponenten der RTSJ zusammen mit einer Bewertung, inwieweit sie hier geeignet erscheinen:

Scheduler Die RTSJ definiert eine Schnittstelle für einsetzbare Scheduler. Es können also prinzipiell beliebige Algorithmen verwendet werden. Der Scheduler verfügt über die Möglichkeit, die Durchführbarkeit von Aufgaben (Tasks) zu prüfen, bevor sie gestartet werden (*Feasibility Analysis*). Dafür wird eine mehr oder weniger genaue Angabe über die Ausführungsdauer benötigt. Für eine Minimalimplementierung schreibt die RTSJ einen Priority Scheduler vor, was nicht gut zu dem hier gewählten und gewünschten Fair-Share Scheduler passt.

Aufgaben Jede Aufgabe unter Echtzeit-Bedingungen wird in einem eigenen `RealtimeThread` oder einer vergleichbaren Klasse gestartet. Das erfordert einen relativ hohen Aufwand bei der Initialisierung und der Synchronisierung.

Parameter Jeder Aufgabe ist ein Parametersatz zugeordnet, die dessen Verhalten bestimmen, dies sind mindestens `SchedulingParameters`, zusätzlich sind u. a. noch `ReleaseParameters` für die Erzeugung von periodischen und sporadischen Aufgaben und `MemoryParameters` definiert. Jede Echtzeit-Aufgabe benötigt diesen Satz von Objekten, der Speicherverbrauch ist für die Verwendung auf kleinen Systemen zu hoch.

Synchronisation Es können verschiedene Algorithmen zur Vermeidung von Prioritätsinversion[22] eingebunden werden. In der Minimalimplementierung wird Prioritätsübertragung (Priority Inheritance) verwendet. Die Synchronisation mit nicht echtzeitfähigen Threads erfordert eine Sonderbehandlung.

Asynchrone Ereignisse Asynchrone Ereignise (Interrupts) werden mittels eines `AsyncEventHandlers` verwaltet, dieser übergibt die zugeordneten Aufgaben beim Eintritt eines Ereignisses an den Scheduler. Mittels `Timer`-Klassen können zeitgesteuerte asynchrone Ereignisse ausgelöst werden. Asynchrone Ereignisse finden keinen Weg in die Yogi2-VM. Sie werden

[22] Prioritätsinversion kann bei gegenseitigen Abhängigkeiten der Threads auftreten. Beispielsweise wenn ein Thread mit einer hohen Priorität auf eine Ressource wartet, die von einem niederprioren Thread bearbeitet wird. Dann kann ein Thread mittlerer Priorität den mit der hohen Priorität blockieren. Prioritätsübertragung ist ein dynamisches Verfahren, um dieses Verhalten auszuschließen. Es gibt auch noch planungsbasierte Verfahren, z. B. das Priority-Ceiling-Protocol [Kee99].

stattdessen bereits im nativen Code und ggf. in Bytecode-Interrupts behandelt.

Speicherverwaltung Die RTSJ spezifiziert verschiedene Speicherbereiche, die hier nicht benötigt werden. `ScopedMemory` verbindet Objekte zu einer Gruppe, so dass sie nach Benutzung gemeinsam bereinigt werden [BCC+03, CC03]. Solange eine Aufgabe nur auf diesen Speicherbereich zugreift, sind keine Unterbrechungen durch die Speicherbereinigung nötig. Da sich die Speicherbereinigung bei der Yogi2-VM hingegen unauffällig verhält (niedrige Ausführungswahrscheinlichkeit), wird ein Echtzeit-Thread nicht von ihm unterbrochen, falls er auf den `new` Operator verzichtet. Ferner definiert die RTSJ noch Zugriff auf physikalischen Speicher, der hier nicht definiert ist (siehe Abschnitt 4.2.4).

Ein der RTSJ ähnlicher aber auch wesentlich schlankerer Ansatz ist in JBed implementiert [Pil02]. Dort wird ein spezieller Thread eingeführt (`Task`), der einen anderen Scheduler verwendet (EDF). Der Parametersatz (Ausführungszeit, Zeitschranke) ist Teil des Threads. Angestoßen wird die Ausführung durch Ereignisse, z. B. mit einem periodischen Timer. Aufgrund der hier vorhandenen starken Speicherplatzbeschränkungen soll ein anderer Ansatz verfolgt werden.

7.2.1 Anforderungen an eine schlanke Echtzeiterweiterung

Ausgehend von obigen Spezifikationen und Implementierungen wird im Folgenden ein Anforderungskatalog für eine Echtzeiterweiterung zusammengestellt:

1. Aufgrund der geringen Speicherplatzverhältnisse soll die Echtzeiterweiterung nach Möglichkeit nur mit einer zusätzlichen Klasse auskommen.
2. Eine Erweiterung mit weiteren anwenderspezifizierten Komponenten wird zunächst nicht vorgesehen. Der Scheduler im Kern der virtuellen Maschine wird entsprechend erweitert, ebenso die internen Thread-Strukturen.
3. Auf eine Abschätzung der Ausführungsdauer einer Aufgabe im Vorfeld der Ausführung wird verzichtet. Diese Abschätzung ist mit geringem Aufwand bei einem dynamischen System wie der JavaVM nur ungenau möglich. Messungen zur Laufzeit sind dabei praktikabler.
4. Aus Vereinfachungsgründen wird auf harte Echtzeitanforderungen verzichtet. Es soll lediglich zur Laufzeit möglich sein, verpasste Zeitschranken festzustellen. Anderenfalls werden die Zeitschranken *genau* eingehalten, es findet keine Drift aufeinanderfolgender Aufgaben statt (Firm-Realtime, [Nil03]).

5. Grundsätzlich kann zwischen statischen und *dynamischen Schedulern* unterschieden werden. Erstere erzeugen den Zeitplan (Schedule) Off-Line, also vor der Programmausführung. Er muss als Parametersatz dem Scheduler übergeben werden und belegt Speicherplatz. Für die Ermittlung des Zeitplans sind die nötigen Ausführungszeiten der Aufgaben sinnvoll, eine Vermischung mit nicht echtzeitfähigen Threads wird schwierig. Dynamische Scheduler ermitteln die Ausführungsfolge der Aufgaben zur Laufzeit (On-Line), der Fair-Share Scheduler ist ein solcher dynamischer Scheduler. Ein guter und einfach zu implementierender Echtzeit-Scheduler, der zudem nicht auf die Ausführungszeiten der Aufgaben angewiesen ist, ist der *Earliest-Deadline-First Scheduler* (EDF). Der zusätzliche Aufwand für den Scheduler (sowohl Programmcode, als auch Rechenzeit) ist gering.

6. Um die Objektzahl zu verringern, soll es möglich sein, einen gewöhnlichen Thread für Echtzeitaufgaben weiterzuverwenden, indem bestimmten Code-Abschnitten eine relative *Zeitschranke* (Deadline) zugeordnet wird. Erreicht ein Thread bei der Ausführung einen solchen Abschnitt, wird der absolute Zeitpunkt der Zeitschranke berechnet und der Scheduler behandelt den Thread entsprechend dieses Zeitpunktes mit EDF. Beim Verlassen des Abschnitts kehrt der Thread wieder in seinen ursprünglichen Zustand zurück. Eine Abfrage der Einhaltung der Zeitschranke erfolgt hier.

7. Es sollen einmalige, periodische und sporadische Aufgaben möglich sein. Auf direkte Unterstützung asynchroner Ereignisse wird verzichtet.

8. Zur Vermeidung der Prioritätsinversion soll *Prioritätsübertragung* stattfinden. Dieses Verfahren wird auch auf Zeitschranken angewendet. Die Prioritäts- und Zeitschrankenübertragung funktioniert für alle blockierenden Vorgänge (Synchronisation sowie Warten auf Speicherbereiningung bzw. Klasseninitialisierung).

Earliest-Deadline-First Scheduling ist ein zeitbasiertes Verfahren das idealerweise in den Scheduler im Kern der JavaVM integriert ist.[23] Die notwendigen Erweiterungen des Schedulers werden in Abschnitt 7.3 zusammen mit dessen grundsätzlicher Funktionsweise beschrieben. Im Folgenden soll zunächst ein Überblick über die Möglichkeiten mit dem implementierten Verfahren, vor allem aus Programmierersicht, gegeben werden.

[23] Viele EDF-Implementierungen, insbesondere solche, die auf Betriebssystem-Threads aufsetzen, verwenden dabei lediglich eine Emulation, die die Zeitschranken zunächst in Prioritäten umsetzt. Ein Prioritry-Scheduler reiht diese dann ein. So kommt es dann zu so obskuren Forderungen nach mindestens 28 Prioritätsstufen [24].

7.2.2 Java-Schnittstelle der Echtzeiterweiterung

Die Idee der hier vorgestellten Echtzeit-Erweiterung liegt darin, Code-Abschnitten eine Zeitschranke zuzuordnen und den Thread für diesen Bereich in einen privilegierten Modus zu schalten. Für die Implementierung von periodischen Aufgaben sollen diese Abschnitte nahtlos verkettet werden können. Hierfür ist eine geeignete Semantik zu finden. In jedem Fall wird für Echtzeit-Aufgaben die übliche phasenweise Bearbeitung verwendet, die sich z. B. auch in [KWK02] bewährt hat:

Vorbereiten Notwendige Initialisierungen für die Aufgabe werden vorgenommen. Dies findet vor der eigentlichen zeitkritischen Sequenz statt. Hier ist es sinnvoll notwendige Objekte anzulegen (ein möglicher Speicherüberlauf, der einen Durchlauf der Speicherbereinigung erfordert findet dann nicht während der zeitkritischen Aufgabe statt). Auch das Initialisieren von Klassen benötigt prinzipbedingt Zeit (siehe Abschnitt 8.1), daher sollten hier zunächst die Klassen, auf die nur statisch zugegriffen wird, einmal verwendet werden.

Durchführen Die eigentliche zeitkritische Aufgabe (Mission) wird bearbeitet. Hier sollten unvorhersagbare Vorgänge vermieden werden, was einfach durch Ausschluss des `new`-Operators erreicht werden kann.

Aufräumen Nach Beendigung der zeitkritischen Sequenz können sich nicht mehr zeitkritische Aufgabenteile anschließen.

Eine naive Implementierung verwendet zur Markierung von privilegierten Code-Abschnitten zusätzliche Methoden in einer Kindklasse von `Thread`, die Zeitschranken setzen und überwachen können. Benutzt werden diese wie die statischen Methoden in `Thread` (`sleep(..)`, `yield()`), die auf den aktuellen Programmfluss wirken:

```
-> prioritätsbasierter Code (Initialisierung)
try {
    EDFThread.setTimeConstraint(200);   // setzt Zeitschranke nach 200ms
  -> EDF-basierter Code, max. 200ms Ausführungsdauer
    EDFThread.endTimeConstraint();      // prüft Einhaltung der Zeitschranke
  -> prioritätsbasierter Code
} catch(TimeViolationException tve) {
  -> prioritätsbasierter Code (Notfallprozedur)
}
```

Der Quelltext-Auszug zeigt den grundsätzlichen Ablauf. Ist dem Thread einmal eine Zeitschranke zugeordnet und im Zustand *Bereit*, so ist er privilegiert und wird vom Scheduler entsprechend des EDF-Schemas vor allen Threads ohne

gesetzte Zeitschranke und vor allen Threads mit einer weniger dringlichen Zeitschranke ausgeführt. Werden im Programmfluss erneut Zeitschranken gesetzt, so werden sie verkettet. Die nach der Berechnung der Aufgabe übriggebliebene Zeit bis zum Erreichen der Zeitschranke wird abgewartet. Dieses Schema funktioniert grundsätzlich, es gibt jedoch ein Konsistenzproblem.

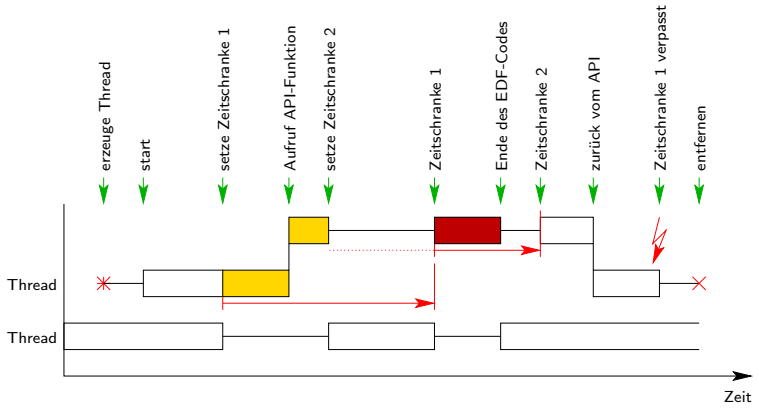


Bild 7.2: Sequentielles EDF: Die verlorene Zeitschranke

Das Beispiel in Bild 7.2 zeigt die Schwachstelle auf. Da die Kontrolle über das Zeitverhalten dem kompletten Programmfluss des Threads erlaubt ist, kann es sein, dass sie nicht vollständig vom Anwendungsentwickler bestimmt wird. In diesem Beispiel wird in der Anwendung eine Zeitschranke gesetzt und dann eine API-Funktion aufgerufen. Für die interne Bearbeitung der Aufgabe setzt die API-Funktion nun ihrerseits eine Zeitschranke, u. U. weiß der Anwendungsentwickler davon gar nichts. Nun gibt es zwei aus Programmierersicht unerwartete Zusammenhänge:

1. In der API-Funktion wird die Ausführung des eigenen EDF-Codes verzögert gestartet, da dieser mit der bereits aktiven Zeitschranke verkettet wird.

2. Die API-Funktion verlässt am Ende den EDF-Zustand und kehrt zurück. Damit ist nicht nur die Zeitschranke der Hauptanwendung verletzt worden, schlimmer noch, die Anwendung läuft nicht mehr im privilegierten EDF-Modus.

Ferner ist es denkbar, dass durch pure Unachtsamkeit des Programmieres ein Thread aus dem zeitkritischen Code-Bereich herausläuft und dennoch im EDF-

Zustand verbleibt, da weder der Compiler noch die JavaVM in der Lage sind, eine paarweise Verknüpfung des Betretens und des Verlassens von EDF-Bereichen zu prüfen.

Eine solche paarweise Verknüpfung von Anweisungen ist in Java bereits eingebaut und wird bei der Synchronisierung angewendet. Der Java-Compiler sorgt dafür, dass die Bytecodes `monitorenter` und `monitorexit` immer paarweise auftreten. Die Lösung ist, EDF-Codesequenzen an synchronisierte Blöcke zu binden. Synchronisiert wird auf ein spezielles *`Deadline`-Objekt*, das in der Lage ist, den vorherigen Zustand des Threads, also die ggf. beim Aufruf vorhandene Zeitschranke, zu sichern. Dieses Verfahren erfordert eine Erweiterung im Synchronisationscode der JavaVM. Nachstehend befindet sich etwas Pseudo-Code mit Java-Syntax für den `monitorenter`-Bytecode, der `monitorexit`-Bytecode wird analog behandelt:

```
monitorenter(someObject) {
    if(someObject.monitorcnt==0) {
        someObject.monitor=currentThread;              // Monitor erwerben
        if(someObject instanceof Deadline) {
            Deadline someDeadline=(Deadline) someObject;
            someDeadline.oldDeadline=currentThread.deadline;
            currentThread.setDeadline(someDeadline.timeConstraint);
        }
    } else if(someObject.monitor==currentThread) {    // derselbe Monitor?
        someObject.monitorcnt++;
    } else {
        currentThread.yield();
    }
```

Da der Zugriff auf das `Deadline`-Objekt und die Weitergabe der Objektreferenz allein dem Programmierer obliegt, können beim Aufruf fremden Codes keine ungewollten Verkettungen mehr auftreten. Die Semantik der Sprache Java zur Synchronisation erzwingt eine entsprechende Strukturierung des Echtzeit-Codes. Im Folgenden werden einige Entwurfsmuster [GHJV95] für die Verwendung des `Deadline`-Objekts in verschiedenen Anwendungsszenarien zusammengestellt.

7.2.2.1 Einmalige Aufgaben

```
try {
    Deadline d=new Deadline(200);
  -> prioritätsbasierter Code
    synchronized(d) {                    // setzt Zeitschranke nach 200ms
      -> EDF-basierter Code, max. 200ms Ausführungsdauer
        d.check();                       // prüft Einhaltung der Zeitschranke
    }
  -> prioritätsbasierter Code
```

```
} catch(DeadlineMissException dme) {
  -> prioritätsbasierter Code (Notfallprozedur)
}
```

Die Monitor-Behandlung der JavaVM wird um eine Abfrage des Objekt-Typs ergänzt. Handelt es sich um den Typ `Deadline`, wird der Modus des Threads entsprechend verändert. Im Code-Beispiel oben wird beim Betreten des Monitors auf das `Deadline`-Objekt `d` auch der EDF-Bereich betreten und beim Verlassen des Monitors wieder zum vorherigen Zustand (hier: normales prioritätsbasiertes Fair-Share-Scheduling) zurückgekehrt. Laut der Definition von synchronisierten Bereichen in Java [LY00] kann dabei keine Ausnahme (Exception) auftreten, daher muss die Zeitschranke mit einer zusätzlichen Anweisung `check()` geprüft werden. Das Verhalten von `check()` entspricht insofern `Object#wait()`, dass es nur auf `Deadline`-Objekte angewendet werden darf, auf die der aktuelle Thread einen Monitor erworben hat, anderenfalls wird eine `IllegalMonitorStateException` erzeugt. Erfordert die Anwendung keine Prüfung der Zeitschranke, so kann es auch entfallen. Der Scheduler führt auch bereits überschrittene Zeitschranken privilegiert aus, es ist also auch möglich, die Notfallprozedur innerhalb des synchronisierten Bereichs mit höchster Dringlichkeit auszuführen:

```
Deadline d=new Deadline(200);
-> prioritätsbasierter Code
synchronized(d) {                       // setzt Zeitschranke nach 200ms
    try {
      -> EDF-basierter Code, max. 200ms Ausführungsdauer
        d.check();                      // prüft Einhaltung der Zeitschranke
    } catch(DeadlineMissException dme) {
      -> EDF-basierter Code (Notfallprozedur)
    }
}
-> prioritätsbasierter Code
```

Das Binden von Echtzeit-Aufgaben an synchronisierte Bereiche ermöglicht es, alle semantischen Konstrukte der Sprache Java, die mit der Synchronisation zusammenhängen, für die Erzeugung und Steuerung von Echtzeit-Aufgaben zu nutzen. Es ist z. B. auch möglich, die Klasse `Deadline` abzuleiten und dort Methoden mit dem `synchronized`-Schlüsselwort zu deklarieren.

7.2.2.2 Periodische Aufgaben

Oft müssen Echtzeit-Aufgaben in regelmäßigen Abständen aufgerufen werden. Dafür könnten dedizierte Zeitgeber-Klassen verwendet werden, aber einfacher ist es, die Zeitverarbeitung des Schedulers selbst dafür zu nutzen. Mittels der

sequentiellen Verkettung von Zeitschranken kann ein exakter periodischer Vorgang modelliert werden. Dafür existiert die Methode `Deadline#append(..)`:

```
try {
    Deadline d=new Deadline(200);
  -> prioritätsbasierter Code
    synchronized(d) {                    // setzt Zeitschranke nach 200ms
        for(;;) {
          -> EDF-basierter Code, max. 200ms Ausführungsdauer
            d.append(200);               // fügt neue Zeitschranke an
        }
    }
} catch(DeadlineMissException dme) {
  -> prioritätsbasierter Code (Notfallprozedur)
}
```

`append(..)` verhält sich zunächt wie `check()`, d. h. die Einhaltung der gerade in Bearbeitung befindlichen Zeitschranke wird geprüft. Im Erfolgsfall verhält sich `append(..)` ähnlich wie ein Verlassen des synchronisierten Bereichs, gefolgt von einem unmittelbaren Wiederbetreten, allerdings ohne zwischenzeitlich den privilegierten Zustand zu verlassen. Die nicht benötigte Rechenzeit der vorherigen Periode wird abgewartet und die neue Zeitschranke wird nicht relativ zum aktuellen Zeitpunkt gesetzt, sondern relativ zur vorherigen Zeitschranke. Innerhalb einer Schleife angewendet lassen sich so periodische Vorgänge ausdrücken. Dieser Mechanismus ist nicht auf äquidistante Zeitschranken begrenzt, wie das folgende Beispiel verdeutlicht:

```
Deadline d=new Deadline(1);
synchronized(d) {                        // setzt Zeitschranke nach 200ms
    for(int x=2; x<=25; x++) {
      -> EDF-basierter Code, max. x-1ms Ausführungsdauer
        try {
            d.append(x);                 // fügt neue Zeitschranke an
        } catch(DeadlineMissException dme) {
          -> EDF-basierter Code (Notfallprozedur)
        }
    }
}
```

Hier wird zu Beginn eine kurze Zeitschranke definiert, die evtl. nicht eingehalten werden kann. Da nachfolgende Zeitschranken aber größere Abstände bekommen, ist es möglich, dass der gesamte Vorgang dennoch innerhalb der geforderten Gesamtzeit abgeschlossen werden kann. Da Zeitschranken sich immer mit der vorhergehenden verketten, auch wenn diese in der Vergangenheit liegen, ist ein Aufholen möglich.

7.2.2.3 Sporadische Aufgaben

Echtzeit-Aufgaben, die nicht durch den Programmfluss des eigenen Threads aufgerufen, sondern durch Ereignisse von anderen Threads angestoßen werden, haben keine fest definierte Periode. Dies ist auch mit den von Java bereit gestellten Mitteln auf ein `Deadline`-Objekt möglich:

```
try {
    Deadline d=new Deadline(200);
    synchronized(d) {                    // setzt Zeitschranke nach 200ms
        for(;;) {
            d.wait();                    // Monitor verlassen, warten
          -> EDF-basierter Code, max. 200ms Ausführungsdauer
            d.check();
        }
    }
} catch(DeadlineMissException dme) {
  -> prioritätsbasierter Code (Notfallprozedur)
}
```

`wait()` bewirkt, dass der Monitor auf das `Deadline`-Objekt freigegeben wird und der aktuelle Thread wartet, bis er von außen mittels `d.notifyAll()` oder `Thread#interrupt()` benachrichtigt wird. Dann wird der Monitor wieder erworben und der Thread setzt seine Arbeit fort. Die Freigabe des Monitors und das Wiedererwerben des Monitors auf das `Deadline`-Objekt hat dieselben Auswirkungen, wie das Verlassen und Wiederbetreten des synchronisierten Bereichs, d. h. es wird auch hier der EDF-Zustand verlassen bzw. die Zeitschranke neu gesetzt. Bei einer Benachrichtigung von außen tritt dieser Thread also unmittelbar in den EDF-Modus ein und bearbeitet die Aufgabe privilegiert entsprechend der durch das `Deadline`-Objekt angegebenen Zeitschranke. Ist die Aufgabe beendet (im nächsten Schleifendurchlauf wird abermals `wait()` aufgerufen), entfällt die Warteperiode, die beim Verketten von Zeitschranken üblicherweise auftritt.

7.2.2.4 Verschachtelte Aufgaben

Durch die Bindung der Echtzeit-Sequenz auf die Erwerbung eines Monitors eines `Deadline`-Objekts kann jedes `Deadline`-Objekt nur einmal gleichzeitig verwendet werden. Ein erneutes Synchronisieren innerhalb der Echtzeit-Sequenz bewirkt kein neues Erwerben des Monitors und hat somit bzgl. des Setzens von Zeitschranken keinen Effekt. Auch eine Verwendung von einem anderen Thread ist durch den gegenseitigen Ausschluss des Monitors nicht möglich. Wohl aber ist es möglich, dass ein anderes `Deadline`-Objekt verwendet wird:

```
try {
    Deadline d=new Deadline(200);
    synchronized(d) {                    // setzt Zeitschranke nach 200ms
      -> EDF-basierter Code, max. 200ms Ausführungsdauer
        synchronized(new Deadline(100)) {
          -> EDF-basierter Code, max. 100ms Ausführungsdauer
        }
      -> EDF-basierter Code, max. 200ms Ausführungsdauer
    }
} catch(DeadlineMissException dme) {
  -> prioritätsbasierter Code (Notfallprozedur)
}
```

Wird hier noch einmal das Szenario vom Anfang des Abschnitts 7.2.2 heran gezogen, so ist keine unbeabsichtigte Verkettung möglich (siehe Bild 7.3), wenn keine Referenz auf die `Deadline` weitergegeben wird, stattdessen werden die Zeitschranken verschachtelt. Natürlich gibt es auch hier keine Garantie, dass

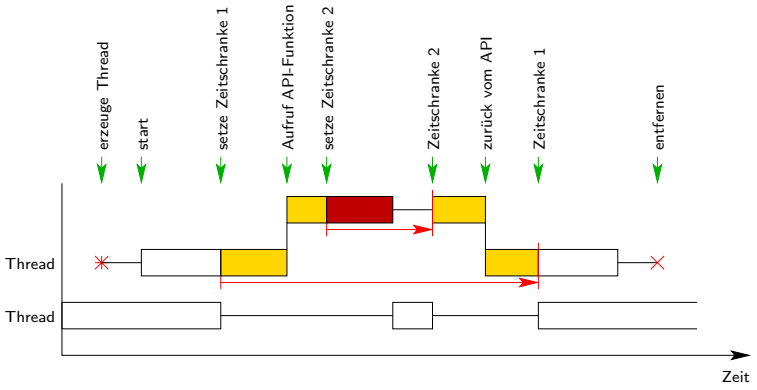


Bild 7.3: Geschachtelte Zeitschranken

alle Zeitschranken eingehalten werden können. Es ist z. B. auch denkbar, dass die API-Funktion eine weniger dringliche Zeitschranke definiert. In diesem Fall muss die umschließende Zeitschranke fehlschlagen.

7.3 Der Scheduler

Die hier beschriebene Echtzeiterweiterung ist eng an die Implementierung der JavaVM gekoppelt. Am deutlichsten wird dies bei der Bindung des EDF-Schedulings an synchronisierte Bereiche. Hierbei sind die VM-internen Routinen

zur Synchronisierung entsprechend anzupassen, dort muss der Typ des zu synchronisierenden Objekts geprüft werden und im Fall eines Objekts vom Typ `Deadline` entsprechend die aktuelle Thread-Struktur manipuliert werden. Dieses Verfahren kann also nicht auf virtuellen Maschinen umgesetzt werden, ohne den Kern zu erweitern. Aber auch der Scheduler selbst hat erweiterte Möglichkeiten und ist als Teil des oben in Abschnitt 7.1 beschriebenen Ablaufs integriert in den VM-Kern. Bereits beschrieben wurde der Umgang mit Threads im Zustand *Wartend*, an dieser Stelle sollen nun Threads im Zustand *Bereit* betrachtet werden.

Zusammengefasst muss der Scheduler folgende Bedingungen garantieren (vgl. hierzu auch noch einmal Bild 7.1):

Dringlichkeit Es handelt sich um einen EDF-Scheduler. Threads mit einer dringlicheren Zeitschranke werden grundsätzlich vor denen mit einer weniger dringlichen Zeitschranke ausgeführt. Die weniger dringlichen Threads befinden sich dann implizit im Zustand *Verdrängt*. Threads ohne gesetzte Zeitschranke werden an letzter Stelle betrachtet (die Zeitschranke liegt dort im Unendlichen). Gibt es mehrere Threads mit derselben Zeitschranke, so werden sie entsprechend der Gleichverteilungs-Bedingung behandelt.

Gleichverteilung Es handelt sich um eine Fair-Share Scheduler. Alle Threads mit derselben dringlichsten Zeitschranke werden entsprechend ihrer gesetzten Java-Priorität mit einer bestimmten Wahrscheinlichkeit (Gerechtigkeit) aufgerufen. Haben Threads dieselbe Priorität, so werden sie im Wechsel bzw. reihum aufgerufen (Round-Robin Scheduling). Für das sich dadurch ergebene dreischichtige Scheduling-Schema (EDF, Fair-Share, Round Robin) gibt es Ausnahmen gemäß den Reaktivitäts- und Blockadefreiheits-Bedingungen.

Reaktivität Grundsätzlich kann jeder Thread an Bytecode-Grenzen unterbrochen werden, falls ein Ereignis dies erforderlich macht. Ein Ereignis hat in jedem Fall einen erneuten Aufruf des Schedulers zur Folge, auch wenn die aktuelle Zeitscheibe noch nicht abgelaufen ist. Threads, die unmittelbar auf ein Ereignis reagieren sollen, werden innerhalb ihrer Dringlichkeitsklasse bevorzugt aufgerufen. Diese Situation tritt immer ein, wenn ein Thread vom Zustand *Blockiert* in den Zustand *Bereit* wechselt. Falls mehrere Threads gleichzeitig aufgeweckt wurden, so werden sie entsprechend ihrer Java-Priorität aufgerufen.

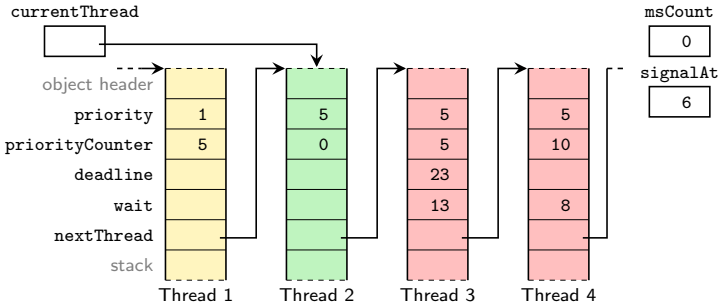
Blockadefreiheit Um das Blockieren von Threads mit einer hohen Dringlichkeit bzw. Priorität durch eine Abhängigkeit zu weniger dringlichen Threads zu vermeiden, werden Prioritäts- und Zeitschrankenübertragung verwendet.

Diese Bedingungen erfüllt ein recht einfaches und leicht zu implementierendes Schema: Jeder Thread verfügt in seiner Datenstruktur, die sich in der Erweiterung des Thread-Objekts befindet, über ein Feld mit einer vermerkten Zeitschranke (`deadline`). Die Zeitschranke kann vom Scheduler mit der mitlaufenden Zeit des Millisekunden-Zählers, der auch die Zeitscheibe des Schedulers bestimmt, verglichen werden. Ferner verfügt jeder Thread über einen Prioritätszähler (`priorityCounter`), der die dynamische Priorität für den Fair-Share Scheduler enthält. Ist ein Thread durch einen anderen Thread blockiert, so wird der blockierende Thread in dessen Struktur zur Nutzung durch den Scheduler vermerkt. Wie schon in Abschnitt 7.1 beschrieben, arbeitet der Scheduler in zwei Durchläufen, in denen die Liste der vorhandenen Threads abgearbeitet wird.

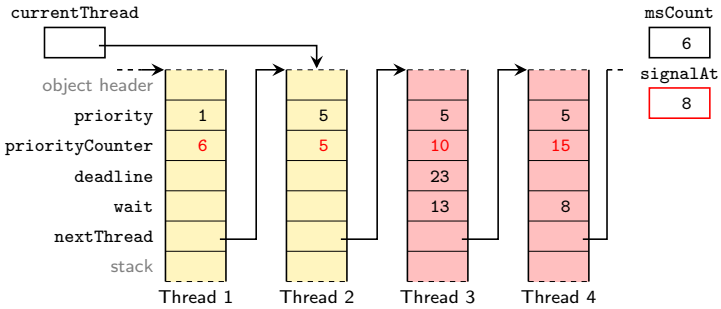
Erster Durchlauf

Es wird eine Bestandsaufnahme aller Threads durchgeführt. Bei jedem Thread im Zustand *Bereit* wird dessen Prioritätszähler um die jeweilige Java-Priorität erhöht. Das Maximum wird dabei für den zweiten Durchgang vermerkt (siehe Bild 7.4 b). Vermerkt wird auch die dringlichste Zeitschranke. Threads im Zustand *Blockiert*, bei denen die Blockadeursache ein anderer Thread ist, übertragen ihre Attribute an den blockierenden Thread, ggf. rekursiv (die Blockade kann durch einen Monitor oder durch eine Abhängigkeit innerhalb der JavaVM bedingt sein, z. B. wenn ein Thread auf die Speicherbereinigung oder den Klasseninitialisierer wartet). Der Prioritätszähler der blockierenden Threads wird zusätzlich um die Java-Priorität des blockierten Threads erhöht (Prioritätsübertragung). Das hat zur Folge, dass der blockierte Thread seine ungenutzte Rechenzeit an den blockierenden Thread abgibt. Bzgl. der Gesamtverteilung der Rechenzeit ist das neutral. Ist bei dem blockierten Thread eine Zeitschranke gesetzt, so wird der blockierende Thread behandelt, als verfüge er über das Minimum beider Zeitschranken (Zeitschrankenübertragung). Die Berücksichtigung der Threads im Zustand *Blockiert* garantiert an dieser Stelle die Blockadefreiheit.[24] Zusätzlich ist es an dieser Stelle möglich, zyklische Blockaden (*Deadlock*s) aufzuspüren und einen Laufzeitfehler der JavaVM auszulösen; dies kann helfen, Programmierfehler in der Anwendung aufzuspüren. Eine Sonderbehandlung erfahren hier noch die Threads im Zustand *Blockiert/Zeit*. In deren Thread-Struktur ist das Ende der Warteperiode vermerkt (gesetzt z. B.

[24] Dieses Verfahren funktioniert mit allen Threads in allen Kombinationen der Scheduling-Parameter. Es muss nicht zwischen Echtzeit- und gewöhnlichen Threads unterschieden werden und es sind keine Sonderbehandlungen nötig. Dies gilt auch für die Speicherbereinigung, fordert ein EDF-Thread (auch wenn er dies nicht tun sollte) Speicherplatz an und dieser ist gerade belegt, so arbeitet auch die Speicherbereinigung mit dessen Zeitschranke, bis der Speicher freigegeben wurde.



a: Thread 2 wird ausgeführt, Aufruf des Schedulers



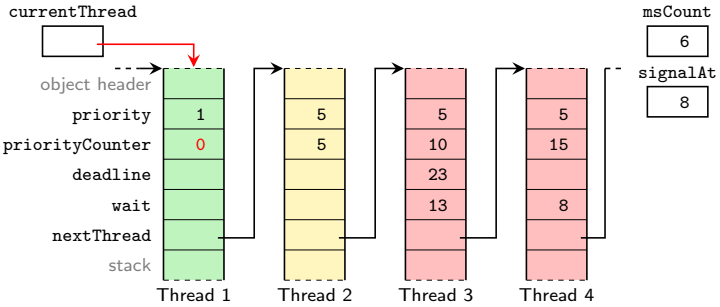
b: Prioritätszähler erhöhen

Bild 7.4: Funktionsweise des Schedulers I

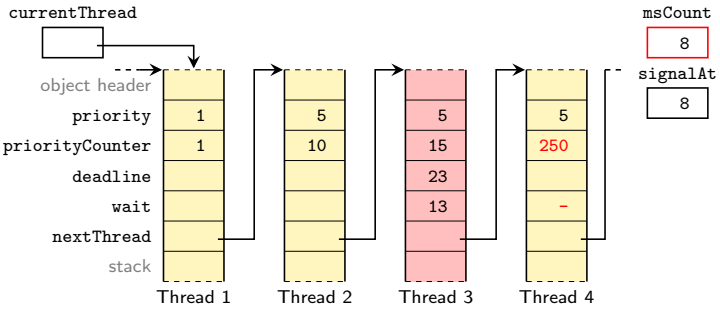
mit `Thread.sleep(..)` oder beim Verlassen einer EDF-privilegierten Code-Sequenz). Der Scheduler erkennt eine abgelaufene Warteperiode und weckt diesen Thread selbstständig auf (in Bild 7.4 b wird dafür ein entsprechender Zeitpunkt für das Unterbrechungssignal des Interpreters eingetragen).

Zweiter Durchlauf

Die zu einen Ring geschlossene Liste der Threads wird durchsucht, um den auszuführenden Thread auszuwählen (falls vorhanden). Es müssen zwei Bedingungen zutreffen: Die gesetzte Zeitschranke des Threads entspricht der ermittelten minimalen Zeitschranke (das erfüllt die Dringlichkeitsbedingung) und der Prioritätszähler des Threads entspricht dem ermittelten maximalen Wert. Da hierbei die Thread-Liste beginnend vom zuletzt ausgeführten Thread durchsucht wird, werden gleichberechtigte Threads reihum aufgerufen. Die beiden



c: Thread 1 wird ausgewählt und ausgeführt



d: Thread 4 wird rechtzeitig aufgeweckt

Bild 7.4: Funktionsweise des Schedulers II

letztgenannten Punkte erfüllen die Gleichverteilungsbedingung. Mittels des Prioritätszählers ist jeder Thread in der Lage, sich ein Guthaben anzusammeln, das seine Ausführungsberechtigung umso mehr erhöht, je länger er nicht aufgerufen wurde. Der Prioritätszähler des entsprechend ausgewählten Threads wird zurückgesetzt und der Thread anschließend aufgerufen (siehe Bild 7.4 c).

Die Bedingung der Reaktivität wird nicht im Scheduler selbst, sondern beim Aufwecken eines blockierten Threads erfüllt. Dort wird der Prioritätszähler des betreffenden Threads einfach auf dessen Maximalwert zuzüglich der eigenen Java-Priorität abzüglich der maximalen Java-Priorität gesetzt. Der Thread wird also innerhalb der Zeitschranken-Klasse mit höchster Wahrscheinlichkeit aufgerufen (siehe Bild 7.4 d).

Dieses Konzept zeigt, dass mit relativ einfachen Strukturen und Algorithmen

auch kleine Java-Systeme mit Echtzeiteigenschaften ausgestattet werden können. Es ergeben sich nur wenige Nachteile. Da die Echtzeiterweiterung in den VM-Kern integriert wurde, ist die Erweiterbarkeit um andere Algorithmen eingeschränkt und bedarf jeweils einer neuen Version der VM. Auf der Java-Seite können keine Erweiterungen definiert werden. Auch harte Echtzeitunterstützung ist in dieser Fassung nicht vorgesehen.

7.4 Leistungsfähigkeit des Schedulers

Dieser Abschnitt soll die prinzipielle Funktionsfähigkeit des hier vorgestellten Konzepts beleuchten. Auf der in Abschnitt 8.3 noch näher beschriebenen Referenzimplementierung der Yogi2-VM wurden einige Messungen durchgeführt.[25] Ein kritisches Element eines Echtzeit-Systems ist die *Antwortzeit* auf Ereignisse (Reaktivität) und dessen Variation bei der Wiederholung (*Jitter*). Die Messungen wurden realisiert, indem der Kern der VM um native Code-Komponenten erweitert wurde, die an bestimmten Stellen der Ausführung den Timer des Mikrocontrollers auslesen. Derselbe Timer erzeugt auch den Millisekunden-Takt. Die ermittelten Zahlenwerte sind auf die Mikrosekunde genau. Natürlich verbraucht der Code für die Messungen seinerseits Rechenzeit. Sie wurde durch eine Code-Analyse zu 48 μs pro Durchlauf bestimmt und aus den Messungen herausgerechnet. Ein kleines Java-Programm wurde geschrieben, das eine Reihe von Tests nacheinander durchführt und die Ergebnisse ausgibt. Jeder Test dauerte drei Sekunden (siehe Anhang E.5.1). Die Tests lassen sich dabei in drei Gruppen einteilen:

1. Mehrere Threads fester Anzahl fordern durchgehend Arrays mit variierender Größe an (Speicherdurchsatz- und -bereinigungs-Stress). Wie in Abschnitt 5.2.3 beschrieben, benutzt die Speicherbereinigung atomare Kopierschleifen für Speicherbereiche. Je größer der zu verschiebende Speicherbereich ist, desto stärker sollte sich dies auf die Antwortzeit auswirken (die Wahrscheinlichkeit eines Ereignisses innerhalb dieses Vorgangs steigt). Beispielsweise benötigt auf dem untersuchten Zielsystem das Kopieren eines Arrays mit 256 Byte-Einträgen ca. 720 μs (ermittelt durch Code-Analyse) und liegt damit bereits in der Größenordnung des Millisekunden-Timers.

2. Die Scheduling-Kosten werden mit derselben Testroutine gemessen, nur wird hier die Anzahl der Threads variiert und die Blockgröße bleibt konstant. Wie im Abschnitt 7.1.1 erwähnt, kann hier wegen der Listenbear-

[25] Erstmals veröffentlicht in [BG04].

beitung beim Scheduling ein Zusammenhang der Bearbeitungszeit (und somit der Antwortzeit) zur Zahl der Threads erwartet werden.

3. Auf der Zielplattform wurden einige teure Routinen implementiert, die mit nativem (also atomarem) Code auf ein grafisches Display zugreifen. Ziel der Messungen soll sein, herauszufinden, wie sich derart grobgranularer Code auf die Antwortzeiten des Systems auf Ereignisse auswirkt.

Bei allen Messungen wird dabei das vom Millisekunden-Timer ausgelöste Schedule-Flag als Ereignis verwendet. Das ist zwar ein internes Ereignis und somit nicht direkt mit einem externen Ereignis vergleichbar, denn nur hiervon kann es ja beobachtbare Antwortzeiten geben, das Signal ereignet sich aber genauso asynchron zum Programmfluss, wie ein externes Ereignis. Der Timer ist eine eigenständige Hardware-Komponente auf dem Controller. Die Ergebnisse dürften durchaus vergleichbar sein. Jeder dieser Tests wurde (sofern sinnvoll und möglich) mit unterschiedlichen Ausführungsparametern durchgeführt:

Running Alle Threads laufen konkurrierend.

Wait/Notify Die Threads rufen (zusätzlich zur Hauptaufgabe) `Object#wait()` und `Object#notifyAll()` im Wechsel auf. Dies testet mögliche Verzögerungen durch den Synchronistionsmechanismus.

Sleeping die Threads rufen (anstatt der Hauptaufgabe) `Thread#sleep(..)` für eine kurze Zeitspanne in einer Schleife auf, warten also die meiste Zeit.

Bild 7.5 zeigt die Messergebnisse. Die vollständigen Werte und Berechnungen sind im Anhang E.5.3 zu finden. Der Graph zeigt die mittlere Reaktionszeit des Systems (die Zeit, die zwischen dem Auftreten des Ereignisses und dem Aufruf des Interpreters für den nächsten Thread vergangen ist) in μs. Die im 1. Test erwarteten Zusammenhänge zwischen der Granularität bei der Ausführung der Speicherbereinigung und der Reaktionszeit haben sich nicht bestätigt. Die Gesamtzeit, die die Speicherbereinigung mit dem Verschieben von Blöcken verbringt ist, verglichen mit der für dem Test benötigen Gesamtrechenzeit, zu kurz. Bei einigen Messwerten gibt es eine hohe Varianz (reproduzierbar). Das kann mit einem Schwebungseffekt (aliasing) beim Aufruf der Speicherbereinigung zusammenhängen.

Eindeutiger sind die Ergebnisse beim zweiten Test. Hier ist gut der vorhergesagte lineare Zusammenhang zwischen Thread-Zahl und System-Reaktionszeit zu erkennen. Auch zu erkennen ist, dass die Synchronisation geringfügig die Reaktionszeit beeinflusst. Das liegt aber nicht am Thread-Scheduler oder -Dispatcher (die Synchronisation wird dort gar nicht beachtet), sondern am Programmfluss. Beim Aufruf von `Object#notifyAll()` wird das Ereignis ausgelöst (impliziert `Thread.yield()`). Es kann jedoch nicht sofort bearbeitet werden,

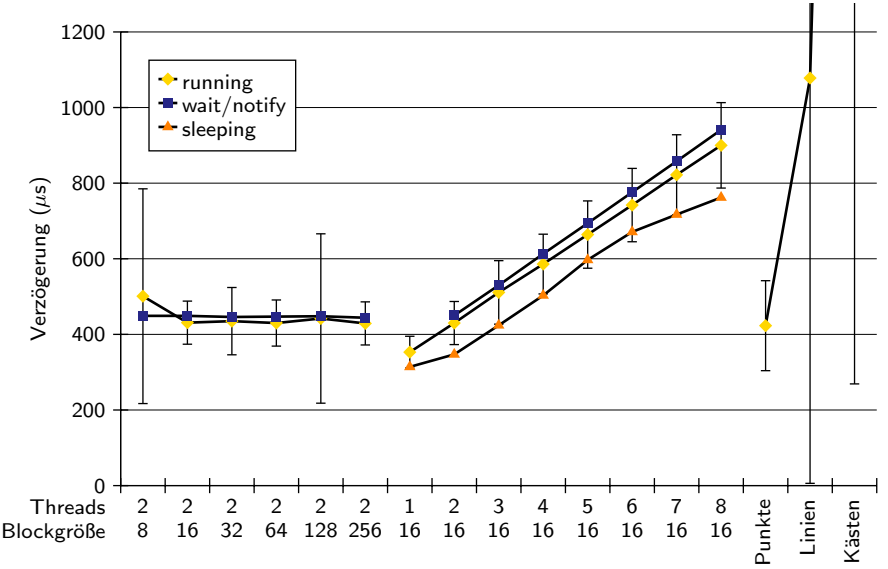


Bild 7.5: Ereignisreaktionszeiten

sondern erst, wenn der ereignisauslösende Thread das `Object#wait()` ereicht hat und den Monitor auf das Objekt freigibt. Ferner ist zu erkennen, dass Threads im Zustand *Blockiert* den Scheduler weniger belasten, als Threads im Zustand *Bereit*. Dort ist nur einer statt zwei Durchläufen über die Thread-Liste nötig. Der etwas andere Programmfluss dieses Durchlaufs (weniger teure Operationen, der Controller wird oft in den Wartezustand versetzt) wirkt sich ebenfalls positiv auf die Reaktionszeit aus. Tabelle 7.1 zeigt die aus dem Messwerten ermittelten durchschnittlichen Verzögerungsquellen der Referenzimplementierung.

Tabelle 7.1: Verzögerungen der Systemzustände

Komponente	**Verzögerungszeit** [μs]*			
	running	*wait/notify*	*sleeping*	mittel
Overhead je Thread	78	81	64	75
Overhead des Schedulers	211	222	249	227

* Jeweils ermittelt aus dem linearen Zusammenhang des zweiten Tests.

Der dritte Test zeigt, dass ununterbrechbare teure native Routinen im Zusammenhang mit Echtzeitanforderungen mit Vorsicht zu genießen sind. Im Fall der gefüllten Rechtecke wurden durchschnittliche Verzögerungen von über 5 ms gemessen. Diese Verzögerungen sind nicht durch das Konzept des Schedulers begründet, sondern liegen ausschließlich an der Rückkehrzeit der Anwendungs-Threads. Darauf kann ein bedachter Programmierer Rücksicht nehmen.

Davon abgesehen zeigen diese Messungen, dass ein 8-Bit-Mikrocontroller in der Lage ist, bis zu acht Threads zu versorgen und dabei beim gewählten Scheduling-Konzept unterhalb einer Millisekunde reagiert, was belegt, dass der gewählte System-Timer von einer Millisekunde dem System angemessen ist. Wenn auf diesem System weniger als vier Threads laufen, kann davon ausgegangen werden, dass das System im statistischen Mittel einer Periode reagiert. Natürlich sind dies nicht die Gesamtreaktionszeiten auf Ereignisse. Hinzu kommt noch die entsprechende Ausführungszeit des Threads, der ausgelöst durch dieses Ereignis seine Aufgabe ausführen soll. Das obliegt dem Anwendungsprogrammierer. In Abschnitt 7.1.1 wurde für normal ausgeführte Threads (also ohne externe Ereignisse) eine Zeitscheibe von 64 ms auf der Referenzimplementierung der VM genannt. Wird dieser Wert hier mit dem Overhead beim Aufruf des Thread-Schedulers/-Dispatchers verglichen, so ergibt sich je nach Threadzahl auf dem System ein Overhead, der im Prozentbereich liegt.

Kapitel 8

Das Gesamtsystem

Wie schon in Kapitel 4 angedeutet, besteht die hier gezeigte Implementierung nicht nur aus einer virtuellen Java-Maschine, sondern umfasst ein komplettes Betriebssystem. Nachdem in den vorhergehenden Kapiteln der Kern der virtuellen Maschine beschrieben wurde, sollen in diesem Kapitel die Komponenten betrachtet werden, die darauf aufsetzen: das Laufzeitsystem, welches für den Betrieb der JavaVM notwendig ist, und eine Programmierschnittstelle (API) für eingebettete Systeme. Ferner wird auf Implementierungsdetails und Erweiterungen auf dem Zielsystem eingegangen.

8.1 Das Laufzeitsystem

Das Laufzeitsystem unterstützt den Mikrokernel der virtuellen Maschine unter Verwendung von Funktionen des Kernels selbst (Speicherverwaltung, Bytecode-Interpreter und Multithreading). Realisiert ist es als ein Java-Thread (*System-Thread*), der eine einzelne statische Methode ausführt. Hierbei werden keine objektorientierten Techniken benutzt und auch keine anderen Methoden aufgerufen, daher kann auf eine Klassenstruktur für diese Methode verzichtet werden.

Der System-Thread fasst alle Aktionen der VM zusammen, die nebenläufig zur Anwendung ablaufen sollen. Das sind im Wesentlichen die Speicherbereinigung und die Klasseninitialisierung. Der Thread läuft mit niedrigster Ausführungswahrscheinlichkeit (Java-Priorität) und ist dementsprechend gleichberechtigt mit den Anwendungs-Threads ohne diese stark zu beeinträchtigen. Die Bearbeitung der Aufgaben wird in Phasen aufgeteilt, die am Ende dieses Abschnitts aufgelistet werden. Jede Phase ist dabei ihrerseits so gestaltet, dass sie nur möglichst kurze Code-Abschnitte enthält, die nicht unterbrochen werden können. Da der System-Thread auf die VM-internen Strukturen zugreifen muss, die von Java aus prinzipbedingt nicht erreichbar sind, besteht er zum größten Teil aus nativem Code. An geeigneten Unterbrechungspunkten wird auf die Bytecode-

Ebene gewechselt, was den Aufruf des Thread-Schedulers/-Dispatchers ermöglicht. Dabei werden dieselben Mechanismen verwendet wie auch für native Methoden der Laufzeitbibliothek (siehe Abschnitt 6.3).

Eine weitere Aufgabe des System-Threads ist die Initialisierung der Anwendung beim Start der VM. Wie schon in Abschnitt 4.2.4 erläutert, wird die Anwendungsklasse ähnlich wie in gewöhnlichen Java-Archiven bestimmt. An dieser Stelle findet aber auch eine Fehlerbehandlung auf der Anwendungsebene statt. Bei schwerwiegenden Fehlern bei der Programmausführung wird ein Fehlercode zwischengespeichert und die VM neu gestartet (siehe Abschnitt 6.4). Stellt der System-Thread an dieser Stelle fest, dass der vorherige VM-Durchlauf mit einem Fehler abgebrochen wurde, so wird statt der `main()`-Methode der Anwendung ein *`ErrorHandler`* initialisiert und die Methode `onError(..)` mit einer entsprechenden Parameterliste mit einer Beschreibung des Fehlers aufgerufen. Auf diese Weise ist es für eine Anwendung möglich, entsprechend auf schwerwiegende Fehler zu reagieren. In jedem Fall wird an dieser Stelle des System-Threads, egal ob es sich um einen gewöhnlichen Anwendungsstart oder eine Fehlerbehandlung handelt, lediglich ein Auftrag zum Starten der entsprechenden Methoden der Klasseninitialisierung übergeben, die Bearbeitung erfolgt dann dort.

8.1.1 Speicherbereinigung

Die bereits in Abschnitt 5.2 ausführlich behandelte Speicherbereinigung läuft als Teil des System-Threads ab. Die vorgestellten Mechanismen zur Prioritäts- und Zeitschrankenübertragung (siehe Abschnitt 7.3) wirken auch auf den System-Thread und somit auch auf die Speicherbereinigung, so dass Threads, die durch nicht ausreichenden Speicherplatz blockiert sind (*Speicherüberlauf*), ihre Rechenzeit auf den System-Thread abgeben. Das hat keine Auswirkungen auf andere, vom Speicherüberlauf nicht betroffene Threads. Die betroffenen Anwendungsthreads werden in den Zustand *Blockiert/Ereignis* versetzt und als auf den System-Thread wartend markiert. Ist ein Speicherbereinigungs-Zyklus abgelaufen bzw. ein kompletter Durchgang des System-Threads mit allen Aufgaben, werden alle derart markierten blockierten Threads wieder aufgeweckt und der System-Thread verliert die zuvor auf ihn übertragenen Privilegien. Die Anwendungs-Threads werden nun erneut versuchen, den gewünschten Speicher anzufordern. Dies kann erneut fehlschlagen, wenn der Bereinigungsfortschritt nicht ausreichend war, so dass sich u. U. dieser Vorgang bis zum Erfolg wiederholt. Dies wird von einem Zähler protokolliert, so dass aussichtslose Speicheranforderungen nach einer bestimmten Anzahl von Versuchen zu einem Abbruch

der Anwendung führen und die VM mit einem *OutOfMemoryError* neu gestartet wird. Diesen Vorgang zeigt Bild 8.1.

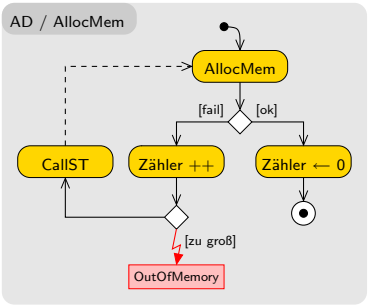


Bild 8.1: Speicherbereinigungsanforderung an den System-Thread

8.1.2 Klasseninitialisierung

Java definiert Klassenlader, die in der Lage sind, von der JavaVM angeforderte Klassen auf dem Zielsystem zu lokalisieren und ggf. über ein Dateisystem oder Netzwerk zu laden. Bei dieser Implementierung wurde auf nachladbare Klassen und austauschbare Klassenlader bewusst verzichtet (siehe Abschnitt 4.2.4). Die Initialisierung einer Klasse ist normalerweise ein aufwändiger Vorgang, bei dem die Klassenstruktur in die für die VM relevanten Zugriffstabellen aufgelöst und eine Verifikation der Bytecodes durchgeführt wird. Dies wird hier allerdings, wie schon in Abschnitt 4.2.5 beschrieben, vor dem Ladevorgang der Klassen auf das eingebettete System durch einen Vorverlinker auf einem Entwicklungssystem erledigt, so dass der Klasseninitialisierungsvorgang der VM selbst recht simpel ausfällt. Dieser besteht lediglich aus der Lokalisation der Klasse entsprechend der Suchreihenfolge innerhalb der Archive und dem Kopieren des Klassenobjekt-Abbildes vom Festwertspeicher auf den Heap. Dieses Abbild enthält bereits die initialisierten Konstanten und die Platzhalter der Laufzeitreferenzen. Praktisch wäre auf diese Weise eine Klasseninitialisierung an den Programmstellen, an denen die Klassen erstmals angefordert werden, möglich, ohne dass dort der Programmfluss insbesondere der von anderen Threads merklich unterbrochen wird. Es könnte zwar umfangreichere statische Klasseninitialisierer[26]

[26] Gemeint ist hiermit die namenlose Methode, die im Java-Quelltext mit dem Schlüsselwort `static{..}` deklariert wurde. Sie erhält beim Übersetzen den Namen `<clinit>`. Die Initialisierung von Klassenvariablen oder zusammengesetzten Konstanten findet ebenfalls dort statt.

geben, diese werden allerdings im Bytecode-Interpreter abgearbeitet und beeinflussen andere Threads dementsprechend nicht. Die Klasseninitialisierung wurde aus mehreren Gründen dennoch im System-Thread implementiert:

Koordinierung paralleler Aufrufe Anwendungen, die aus mehreren Threads bestehen, könnten u. U. dieselbe Klasse mehrfach initialisieren, wenn ein Threadwechsel bei der Initialisierung einer Klasse erfolgt und diese noch nicht zu Verfügung steht. Das umfasst auch die nötige Initialisierung der Superklassen. Um dies zu verhindern, müssen alle in der Initialisierung befindlichen Klassen zentral verwaltet werden. Sie dann auch zentral zu initialisieren, ist ein logischer Schritt der Vereinfachung.

Erweiterbarkeit In dieser Implementierung der Yogi2-VM ist es zwar nicht vorgesehen, in weiteren Implementierungen kann aber sehr wohl die vollständige Analyse einer Klassendatei vorgesehen werden, ohne einen Vorverlinker einzusetzen, z. B. um Klassen aus externen Quellen zu laden. In diesem Fall muss die Initialisierung in verschiedene Phasen unterteilt werden.[27]

Zusatzfunktionen In die Klasseninitialisierung wurden einige Zusatzfunktionen integriert, um Code zu zentralisieren. Es kann eine Methode mit einem Parametersatz angegeben werden, die nach der Initialisierung der Klasse gestartet werden soll, wahlweise in einem neuen Thread. Auf diese Weise wird beispielsweise die Anwendung beim Start der VM aufgerufen.

Praktisch gelöst wurde die Koordinierung der Klasseninitialisierung mittels Objekten, die einen *Initialisierungsauftrag* kapseln, den `Init`-Blöcken. Möchte ein Anwendungsthread auf eine Klasse zugreifen, die nicht in der Laufzeitreferenzentabelle der aktuellen Klasse vermerkt ist, so wird sie zunächst im Heap symbolisch (über ihren Namen) gesucht und bei Erfolg in der Laufzeitreferenzentabelle eingetragen und benutzt. Wurde die Klasse nicht gefunden, wird der Heap nach einem passenden `Init`-Block durchsucht. Bei Erfolg wird der Thread genauso, wie es bei einem Speicherüberlauf geschieht, in den Zustand *Blockiert/Ereignis* versetzt und auf den System-Thread wartend markiert. Wurde kein passender `Init`-Block gefunden, wird er neu angelegt und der erzeugende Thread dort eingetragen. Er kann dann ggf. dazu benutzt werden, den statischen Klasseninitialisierer auszuführen.

Der System-Thread führt nach Abschluss eines Speicherbereinigungs-Zyklus die Klasseninitialisierung auf dem kompaktierten Heap durch, denn dann ist

[27] In der Tat verfügte eine frühe Version dieser Implementierung über einen solchen Mechanismus, es hat sich jedoch gezeigt, dass der nötige Heap-Speicherplatz für die zur Laufzeit erzeugten Taballen (CPR, MRT, siehe Abschnitt 4.2.3) zu klein ist. Daher wurde dieser Weg auf diesem Zielsystem nicht weiter verfolgt und hierfür kein Code eingebunden.

die Wahrscheinlichkeit eines Speicherüberlaufs am geringsten. Der Heap wird nach `Init`-Blöcken durchsucht und sie werden nacheinander bearbeitet. Sind für einen Initialisierungsauftrag nicht alle Voraussetzungen erfüllt (z. B. wenn die Superklasse noch nicht vollständig initialisiert ist oder nicht genügend Speicherplatz für das Klassen-Objekt auf dem Heap zur Verfügung steht), so wird er zunächst ausgelassen und noch einmal im nächsten Durchgang des System-Threads bearbeitet. Dadurch kann in dem einen System-Thread eine quasi-nebenläufige Abarbeitung aller `Init`-Blöcke erreicht werden. Zur Koordinierung verfügt jeder `Init`-Block über ein Zustandsregister, das den auszuführenden *Initialisierungsschritt* enthält:

- Superklasse lokalisieren,
- Größe bestimmen und Speicher anfordern,
- Klassenabbild kopieren,
- ggf. Thread erzeugen,
- den statischen Klasseninitialisierer und ggf. die gewünschte zu startende Methode einem Thread zuschlagen[28] und
- den `Init`-Block vom Heap entfernen.

Die `Init`-Blöcke werden nicht auf dem Heap referenziert, sie werden daher von der Speicherbereinigung ausgenommen, dasselbe gilt für neu erzeugte Klassen-Objekte, welche möglicherweise erst später eine Referenz bekommen (der anfordernde Thread wird ja nur geweckt, es könnte jedoch sein, dass er nicht als nächster vom Scheduler ausgewählt wird, erst wenn er tatsächlich ausgeführt wird, findet er das Klassen-Objekt und erzeugt eine Referenz). Die Vorgänge bei der Klasseninitialisierung werden in Bild 8.2 noch einmal verdeutlicht, dort ist auf der rechten Seite die Bearbeitung der Initialisierungsschritte als Zustandsautomat (Nutzfall-Lebenszyklus) modelliert, mit den optionalen Erweiterungen. Er ist eingebettet in ein Ablaufdiagramm, welches die Abfolge der Bearbeitung der einzelnen Schritte aller `Init`-Blöcke darstellt.

[28] Es wird einfach ein `invokestatic`-Bytecode nachgebildet und der entsprechende Rahmen dem gewünschten Thread hinzugefügt. Da es dabei keine Parameter und Rückgabe-Werte gibt, hat dies keine Auswirkungen auf die dort laufende Methode (falls vorhanden). Sie wird nach Beendigung der hinzugefügten Methoden an der Stelle fortgesetzt, an der sie zuvor unterbrochen wurde.

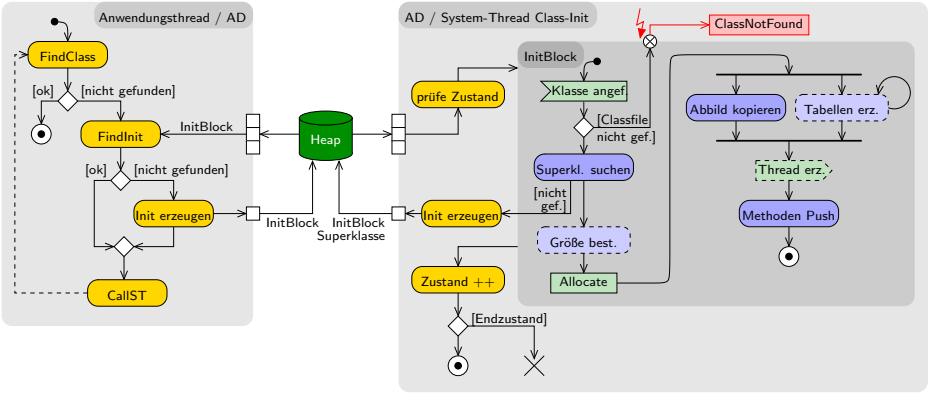


Bild 8.2: Klasseninitialisierungsaufträge an den System-Thread

8.1.3 Die System-Thread-Phasen

Der System-Thread hat also vielschichtige Aufgaben, die hier der Übersicht halber in Phasen aufgeteilt und noch einmal zusammenfassend aufgezählt werden:

Start Die Anwendung oder der entsprechende Fehlerbehandler werden zur Initialisierung vermerkt. Das passiert einmalig beim Start der VM, die übrigen Phasen werden in einer Schleife ausgeführt.

Reset Das Heap-Dirty-Flag wird zurückgesetzt (Erklärung folgt unten).

Root-Set Alle Objekte vom Typ `Thread`, die sich nicht im Zustand *Beendet* befinden, werden grau gefärbt, alle anderen weiß, hierzu wird der Heap einmal abgearbeitet ($O(n)$ mit n=Anzahl aller Blöcke).

Mark Die Speicherbereinigung untersucht alle grauen Blöcke nach Handle-Referenzen und färbt die referenzierten Blöcke ebenfalls grau; nach Abarbeitung aller Referenzen eines Blocks wird dieser schwarz gefärbt, der Heap wird mehrfach abgearbeitet, bis nur noch schwarze und weiße Blöcke übrig sind ($O(n)$ mit n=Anzahl der benutzten Blöcke).

Sweep Die Speicherbereinigung entfernt alle Objekte mit weißer Färbung vom Heap, hierzu wird der Heap ein weiteres mal abgearbeitet ($O(n)$). Auf die Bearbeitung von Objekt-Finalisatoren (`finalize()`-Methoden) wird in der Yogi2-VM verzichtet, dazu wäre ein weiterer eigenständiger Thread nötig.

Compact Die Speicherbereinigung schiebt die übrig gebliebenen Blöcke zusammen. Hierfür stehen z. Zt. zwei Implementierungen zur Verfügung (vgl. Abschnitt 5.3). Bei der sequentiellen Suche muss der Heap jeweils nach dem Block mit der niedrigsten Adresse durchsucht werden, anschließend nach dem Block mit der nächst höheren. Das ist der kostenintensivste Teil des System-Threads mit einer quadratischen Abhängigkeit von der Anzahl der benutzten Blöcke ($O(n^2)$). Alternativ steht eine Implementierung mit einer Listenverwaltung zur Verfügung, die auf Kosten der Code-Größe nur eine lineare Abhängigkeit des Aufwands zur Blockzahl bietet ($O(n)$). Ist zwischen den beiden aufeinander folgenden Blöcken eine Lücke (ein ehemaliger weißer Block), so wird der höhere Block umkopiert, so dass er an den unteren anschließt. Der im Handle vermerkte Zeiger wird angepasst. Umkopieren und Anpassen des Zeigers sind dabei atomare Vorgänge. Dieses wiederholt sich mit den Blöcken der jeweils nächsthöheren Adressen, bis alle Blöcke abgearbeitet sind.

Free Der freie Bereich wird mit Null-Werten gefüllt, so dass später neu allozierte Blöcke mit definierten Werten gefüllt sind ($O(m)$ mit m=Größe des freigegebenen Bereichs). Das Füllen nicht bei der Allozierung, sondern hier durchzuführen, ist effektiver (nur ein Schleifendurchlauf). Die spätere Allozierung von Blöcken selbst schließt dann schneller ab (reaktiveres Verhalten). Anschließend wird der Zeiger auf den freien Bereich (Nursery) angepasst.

Init Der Heap befindet sich jetzt in dem Zustand höchster Kompaktheit. Das erhöht die Wahrscheinlichkeit, dass dieser Schritt erfolgreich abschließt. Nun werden Klassen initialisiert, der Heap wird nach Blöcken vom Typ `Init` durchsucht und bei jedem gefundenen wird ein Initialisierungsschritt ausgeführt. Jeder `Init`-Block speichert dazu einen Zustand. Diese Aufteilung war nötig, da bei der Klasseninitialisierung untereinander Abhängigkeiten bestehen. Kann aufgrund dieser Abhängigkeiten oder mangelndem Speicherplatz ein Initialisierungsschritt nicht durchgeführt werden, so wird er zurückgestellt und beim nächsten Durchlauf des System-Threads erneut aufgerufen.

Wake-Up Auf den System-Thread wartende Threads (erkennbar am entsprechenden Flag im Thread) werden aufgeweckt, egal ob die Ursache fehlender freier Heap-Speicher oder eine fehlende Klasse war. Die entsprechenden Threads werden nun an der Stelle fortgesetzt, an der sie angehalten wurden, und die Bedingung, die zum Blockieren geführt hat, erneut geprüft. Dieser Vorgang ist wesentlich einfacher, als im System-Thread eine Liste der auf ihn wartenden Threads zusammen mit ihren Anforderungen zu führen.

Check Das Heap-Dirty-Flag wird geprüft. Ist es gesetzt, so startet der System-Thread unmittelbar in der Reset-Phase erneut, andernfalls wird der System-Thread zunächst in den Zustand *Blockiert/Ereignis* versetzt.[29] Das Heap-Dirty-Flag wird bei der Allozierung von Blöcken, bei deren Freigabe, beim Beenden von Threads und bei noch nicht vollendeter Klasseninitialisierung gesetzt. Ist der System-Thread dabei im Wartezustand, so wird er wieder aufgeweckt.

Sollte in der Mark-Phase festgestellt werden, dass nur noch ein Block auf dem Heap vorhanden ist (der System-Thread selbst), so wurden alle Anwendungs-Threads beendet und sämtlicher Speicher freigegeben. In diesem Fall beendet sich die VM regulär und startet neu.

Bild 8.3 zeigt das zeitliche Zusammenspiel zwischen den Anwendungsthreads und des System-Threads bei Speicheranforderungen und Klasseninitialisierungsaufträgen.

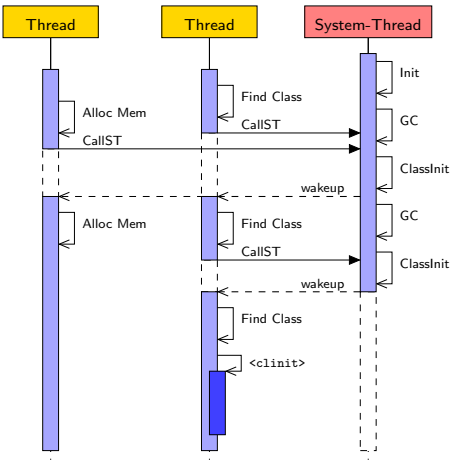


Bild 8.3: Der System-Thread

[29] Dies ist unbedingte Voraussetzung dafür, dass der Mechanismus zum Schlafenlegen des Prozessors, um den Stromverbrauch des Systems zu senken, funktioniert (siehe Abschnitt 7.1.1).

8.2 Eine API-Bibliothek für Embedded Control

Bereits in Kapitel 3 wurde das hier vorgestellte Java-Konzept mit anderen Java-Varianten verglichen. Dabei spielte nicht nur die Komplexität der virtuellen Maschine, sondern vor allem auch die eingesetzte *Programmierschnittstelle* eine entscheidende Rolle. Sie muss nicht nur in der Größe zum Zielsystem passen, sondern auch ein entsprechendes Einsatzgebiet abdecken. Hier soll es um die Steuerung und Regelung von Geräten gehen sowie um die Kommunikation über einfache Netzwerke. Da dies von vorhandenen Bibliotheken nicht abgedeckt wird, wurde mit dem *JControl-API* [25] eine speziell für diesen Einsatzzweck angepasste Programmierschnittstelle eingeführt, die bei der Firma `domo:logic` unter Beteiligung dieser Arbeit entstand. Es wurde insbesondere auf die Rahmenbedingungen, welche sich aus der Umgebung (8-Bit-Mikrocontroller) ergeben, geachtet und das API entsprechend angelegt. Gegenüber dem Szenario der Programmierschnittstelle in Abschnitt 1.2.2 der Einleitung hat sich eine ähnliche Struktur manifestiert:

8.2.1 Hierarchische API-Strukturen

Die Programmierbibliothek ist auf diesem Zielsystem grundsätzlich in zwei Teile aufgeteilt, einen fest auf dem Controller integrierten und in einen nachladbaren. Ein weiteres Kriterium zur Aufteilung sind die Grundfunktionen von Java und spezielle Klassen zur Ansteuerung von Hardware. Die Bibliothek wurde daher in drei Schichten angeordnet:

Systemkomponenten Ein minimales Subset der Java-Laufzeitbibliothek (J2SE) sowie eigene Spracherweiterungen ermöglichen die Grundfunktionalität von Java, z. B.:
`Object`, `Class`, `String`, `Management`, `Deadline`.

Interne Komponenten Zugriff auf die Peripheriekomponenten des Controllers sowie daran angeschlossene Hardware wird ermöglicht (nativer Code wird verwendet), z. B.:
`GPIO`, `PWM`, `ADC`, `Keyboard`, `RS232`, `I2C`, `RTC`, `CAN`, `Display`.

Externe Komponenten In den externen Flash-Speicher werden die Anwendung und bei Bedarf weitere Bibliotheken geladen. Ein direkter Zugriff auf die Controller-Hardware ist von dort nicht möglich (nativer Code kann nicht verwendet werden), z. B.:

- Grafische Benutzeroberflächen (Vole, Viper),
- Zugriff auf spezifische I^2C-Chips und den SM-Bus,

- höhere Netzwerkprotokolle u. a. für die Home-Automation, z. B. CAN, EIB (European Installaton Bus), Konnex.

Mittels dieser API-Anlage wurde ein flexibles System geschaffen, das Produktvarianten durch Anpassung der internen Komponenten und vielseitige Anwendungen durch entsprechendes Nachladen externer Komponenten erlaubt. Die externen Komponenten sind mittlerweile recht umfangreich geworden, sie wurden zu einem großen Teil bei der Firma `domo:logic` entwickelt und sollen im Rahmen dieser Arbeit nicht weiter betrachtet werden. Bild 8.4 zeigt eine Anwendung, die unter Benutzung dieser Programmierbibliothek entstanden ist. Die beiden Screenshots zeigen die Kombination der grafischen Benutzeroberfläche Vole mit der I^2C-Bibliothek zur Visualisierung von Messwerten eines PCs.

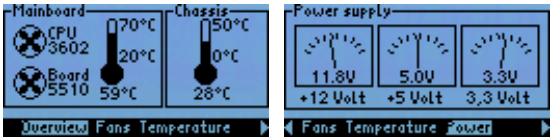
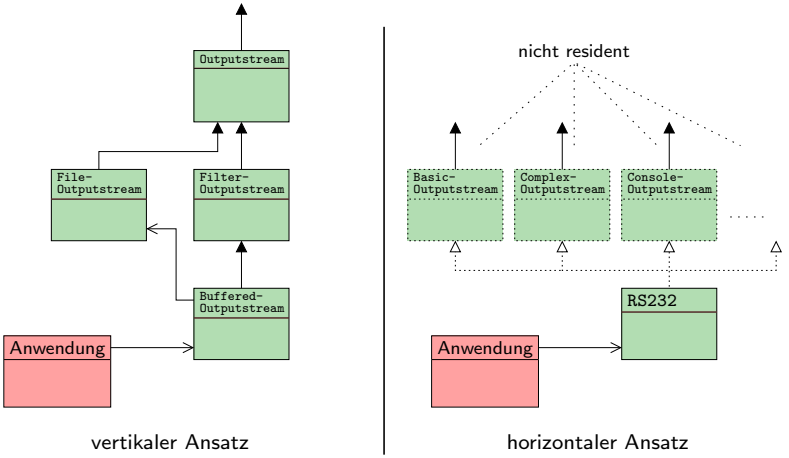


Bild 8.4: *JControl* Beispiel-Screenshots

8.2.2 Horizontale Vererbung

Nicht nur die Grobstruktur wurde auf das Anwendungsprofil ausgelegt, sondern auch die Feinstruktur auf das Speicherprofil. Ziel musste auch hier bei möglichst großer Funktionalität auf der Anwendungsebene ein möglichst geringer Ressourcenbedarf auf der VM-Ebene sein. Hierfür haben sich die Strukturen, die durch die J2SE vorgegeben wurden, als ungeeignet herausgestellt, da für eine bestimmte Basisfunktionalität zuviele Klassen geladen werden müssen. Als ein Grund dafür hat sich eine große Vererbungs- und Abhängigkeitshierarchie von Klassen gezeigt. In [BT02] wurde dies mittels eines Beispiels von I/O-Streams demonstriert (siehe Bild 8.5). Um eine bestimmte Datenstrom-Funktionalität zu erreichen (z. B. `println(..)`) müssen zunächst eine Reihe von abhängigen Streams initialisiert werden, die weitere Funktionalitäten bereitstellen. Nicht nur die dazu benötigten Klassen und Stream-Instanzen belegen auf eingebetteten Systemen wertvollen Heap-Speicherplatz, sondern es belegt auch die sequentielle Bearbeitung der Datenströme je Instanz einen eigenen Datenpuffer.

Um die Anzahl der nötigen Klassen und Instanzen zu verringern, werden in diesem API-Ansatz Interfaces verwendet. Diese bieten einen weiteren Verer-



Erstmalig publiziert in [BT02].

Bild 8.5: Vertikale vs. horizontale Vererbung

bungsmechanismus, der *horizontal* ausgelegt ist. Interfaces selbst sind genauso hierarchisch organisiert wie gewöhnliche Klassen. Wenn aber eine Klasse ein oder mehrere Interfaces implementiert, so wird immer nur Bezug auf die jeweils höchste Hierarchiestufe genommen. Entscheidend bei diesen Konzept ist die Verwendung eines Vorverlinkers (siehe dazu die Methodenreferenztabelle der Interfaces in Abschnitt 4.2.3.2). Dabei wird die Hierarchie der Interfaces aufgebrochen und in die implementierenden Klassen integriert. Die Interface-Klassen selbst müssen auf dem Zielsystem nicht vorhanden sein. Entsprechend der gewünschten Funktionalität können dann die Klassen des APIs die passenden Interfaces implementieren, nicht vorhandene (da nicht sinnvolle) Kombinationen werden dann bei der Anwendungserstellung durch den Compiler erkannt.

Der Nachteil dieses Verfahrens ist allerdings, dass die Interface implementierenden Klassen selbst den Code der Realisierung bereitstellen müssen, es kann also ggf. zu mehrfachem mehr oder weniger identischen Code kommen, der sich im API verteilt. Hierbei wird also die Ersparnis wertvollen Heap-Speicherplatzes durch einen Mehrverbrauch von etwas weniger knappen Code-(ROM-) Speicherplatz ersetzt. Es ist eine entsprechende Balance zu finden. Geringe Möglichkeiten zur Zusammenlegung von Code bieten sich auf der Ebene des nativen Codes, wobei Klassengrenzen aufgebrochen werden können. Ei-

ne Bibliothek von Unterroutinen stellt die spezifizierten Funktionalitäten der Interfaces bereit.

Bei der J2ME [7] wurde mit dem Connection-Framework die Programmierung von I/O- und Netzwerk-Funktionalitäten vereinfacht. Es wird ein Framework von Connection-Klassen bereit gestellt. Sie werden über eine gemeinsame Factory-Methode mittels URL-Parameter-Strings ausgewählt und geöffnet. Hierbei wurden auch die vom Programmierer benötigten I/O-Streams auf je einen Satz von `xxxStream` und `DataxxxStream` reduziert. Nicht aufgebrochen wurde jedoch deren Hierarchie, so dass je Datenstrom wiederum recht viele Klassen resident sein müssen. Dieser Ansatz wurde daher auf diesem Zielsystem nicht weiter verfolgt, auf etwas größeren Systemen erscheint er jedoch interessant.

8.3 Implementierungsdetails

Schon mehrmals wurde auf die Referenzimplementierung der Yogi2-VM auf dem ST7-Mikrocontroller Bezug genommen. An dieser Stelle soll etwas genauer auf die Eigenheiten dieser speziellen Implementierung eingegangen werden. Sie war auch meist die Grundlage der hier vorgestellten Forschungsergebnisse und Messungen.

8.3.1 Zielsystem ST7

Der ST7 von ST-Microelectronics [26, 27] ist ein typischer 8-Bit-Mikrocontroller und sehr preiswert in vielen Varianten am Markt erhältlich. Der ST7 verfügt über 64 KByte Adressraum in Von-Neumann-Architektur. Je nach Variante stehen als Laufzeitspeicher bis zu 2 KByte statisches RAM und als Festwertspeicher bis zu 60 KByte OTP-ROM zur Verfügung (siehe Bild 8.6). Für die Realisierung der JavaVM sind nur die Varianten der größten Ausbaustufe (ST72311R9 bzw. ST72511R9) geeignet. Tatsächlich steht dort sogar noch etwas mehr Laufzeitspeicherplatz zur Verfügung, als offiziell in den Datenblättern angegeben: Der als reserviert bezeichnete Bereich von `0880h` bis `0BFFh` ist ebenfalls mit statischem RAM belegt. Ferner wird der Bereich von `0080h` bis `013Fh` für VM-interne Variablen verwendet und der Bereich `0140h` bis `017Fh` als ST7-Stapelspeicher benutzt (davon wird nur sowenig benötigt, da die Java-eigene Speicherverwaltung ausschließlich den Heap benutzt). Von insgesamt 2944 Bytes Laufzeitspeicher stehen folglich 2688 Bytes für den Java-Heap zur Verfügung.

Großer Vorteil der ST7-Familie sind die zahlreichen Peripheriekomponenten auf dem Chip (siehe Bild 8.7): Frei programmierbare Ein-/Ausgabepins sind

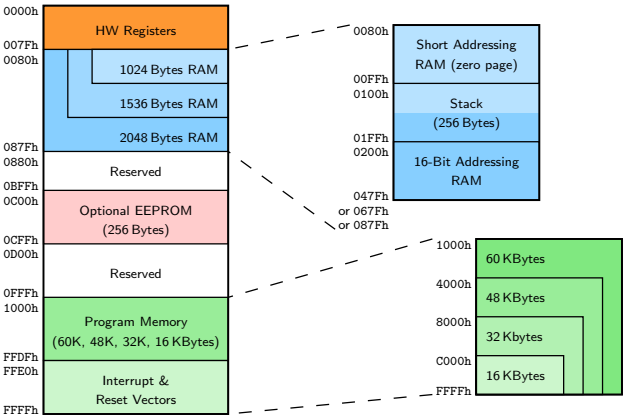


Bild 8.6: Speicheraufbau des ST7-Mikrocontrollers (aus [27])

je nach Ausstattungsvariante auf bis zu sechs Ports zu je acht Pins verteilt. Einige der Pins können mit Sonderfunktionen belegt werden. Dazu zählen eine programmierbare Pulsweitenmodulation (Bestandteil eines 8-Bit-Timers; 4 Pins) und ein Analog-Digital-Wandler (10 Bit; 8 Pins). Einige standardisierte serielle Schnittstellen (SCI bzw. RS232, SPI bzw. I^2C und in einigen Varianten auch CAN) verwenden optional weitere Pins. Die meisten dieser Schnittstellen stehen unter Java zur Verfügung und werden mit dem *JControl*-API angesteuert (siehe Abschnitt 8.2.1). Schließlich existieren noch zwei programmierbare 16-Bit-Timer, von denen einer für die interne Zeitsteuerung des Thread-Schedulers verwendet wird, sowie ein Watchdog-Timer, der im Fall eines VM-internen Fehlers den Controller automatisch neu startet. Ferner wird der Controller durch eine integrierte Spannungsüberwachung (LVD) gesteuert, so dass keine fehlerhaften Programmausführungen wegen zu niedriger Betriebsspannung möglich sind. Alle Peripheriekomponenten werden über den Speicherbereich `0000h` bis `007Fh` angesteuert (Memory-Mapped I/O).

Die Taktfrequenz des ST7-Rechenkerns liegt bei maximal 8 MHz (der externe Takt beträgt 16 MHz), dabei werden ca. 2 MIPS eines einfachen CISC-Befehlssatzes erreicht. Die typische Leistungsaufnahme beträgt bei 3,3 V ca. 50 mW unter Vollast und 15 mW im Ruhezustand. Der Ruhezustand wird von der VM in Bearbeitungspausen aller Threads mit der `wfi`-Anweisung eingeleitet. Weitere Energiesparmaßnahmen sind zwar technisch möglich, beeinflussen aber auch die Peripheriekomponenten auf dem Chip, so dass von deren Verwendung bei der Yogi2-VM abgesehen wurde.

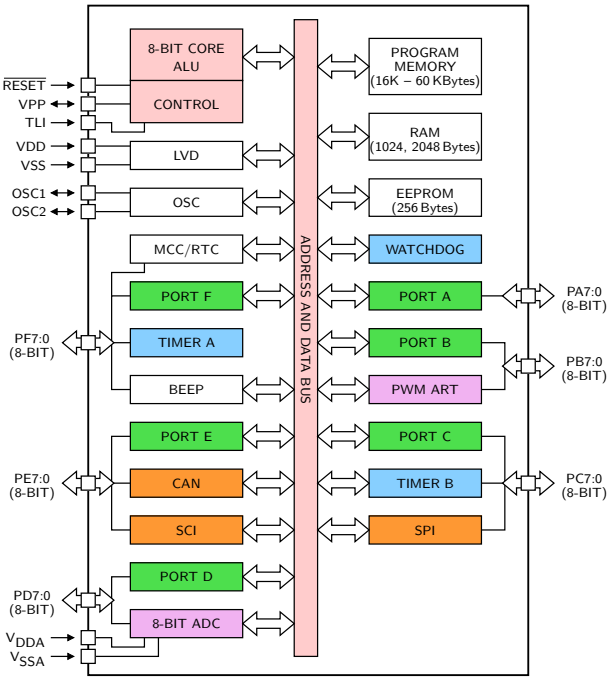


Bild 8.7: Blockdiagramm des ST72511-Mikrocontrollers (aus [27])

Dass die Wahl des Zielsystems in diesem Projekt auf den ST7 gefallen ist, hatte weitestgehend historische Gründe wie Verfügbarkeit von Entwicklungswerkzeugen und Erfahrungen. Erst später hatte sich gezeigt, dass diese Architektur für Java nur relativ geringe Rechenleistung bietet (siehe Abschnitt 6.5).

8.3.2 Realisierung

Implementierungssprache für den Kern der virtuellen Machine und nativen Code ist Assembler. Diese Sprache wurde wegen der besseren Möglicheiten, den Speicherverbrauch des Codes zu beeinflussen, gewählt. Es hat sich gezeigt, dass Assembler gegenüber der Sprache **C**, die für den ST7 ebenfalls verfügbar ist, weitere Vorteile bietet:

Keine Abhängigkeit von den Datenstrukturen einer Hochsprache C und andere Hochsprachen haben ein recht starres Konzept, mit dem die Programmstrukturen auf die Maschinenebene umgesetzt werden. Beim Prozeduraufruf erfolgt die Parameterübergabe beispielsweise über den Call-Stack des Controllers. Da der ST7 nur über einen nativen Stack verfügt, können Java-spezifische Konstrukte nur schwer umgesetzt werden, so würden z. B. beim Multithreading umfangreiche Kopieroperationen anfallen.

Eigene Speicherverwaltung Um lokale Variablen zu ermöglichen, wird in **C** ein eigener Heap benötigt, der mit dem Java-Heap konkurrieren würde, auch dann, wenn keine Rekursion verwendet wird und die lokalen Variablen statisch angelegt werden.

Programmiertricks Schließlich bietet Assembler viele Möglichkeiten, den Programmfluss zu gestalten und zu beeinflussen, die über die Fähigkeiten einer Hochsprache hinausgehen. Das wurde beispielsweise in der Hauptschleife des Interpreters (Bytecode-Sprungtabelle) oder bei der Umschaltung zwischen Java-Bytecode und nativen Code angewendet (siehe Abschnitt 6.3).

Natürlich hat die Verwendung von Assembler auch Nachteile. Neben nur umständlichen Möglichkeiten Algorithmen zu beschreiben und fehlenden Datentypen (auf dem Zielsystem existieren nur 1- und 8-Bit-Datentypen, höherwertige müssen zusammengesetzt werden) ist es vor allem die freie – oder anders ausgedrückt: fehlende – Strukturierbarkeit. Um dem zu begegnen, wurden für dieses Projekt Strukturen und Richtlinien definiert, welche an dieser Stelle nur kurz angerissen werden sollen:

Projektaufteilung Der native Code der virtuellen Maschine besteht aus zwei Teilen, Code für den Kern, der sich in einem Verzeichnis befindet, und Code für native Methoden, der sich über einzelne Dateien verteilt. Deren Namen und Pfad ergeben sich entsprechend der (vollständigen) Bezeichnung der Java-Klassen, in denen die jeweiligen Methoden mit dem `native`-Schlüsselwort deklariert wurden. Der Code für den Kern ist in 16 Module gegliedert. Der Vorverlinker wurde entsprechend erweitert, so dass er aus dem Bytecode und nativen Code der Laufzeitumgebung ein weiteres Modul zusammenstellt. Alle Module zusammen werden in ein Binär-Abbild assembliert. Diesen Vorgang illustriert Bild 8.8.

Sichtbarkeit Assembler lässt keine wirkliche Zugriffskontrolle auf Code-Sequenzen und Variablen zu. Sie wird daher über eine Sichtbarkeitskontrolle nachgebildet, die auf Modulgrenzen funktioniert. Dazu verfügt jedes Modul über eine Liste zu exportierender Symbole. Weiterhin löst der Vor-

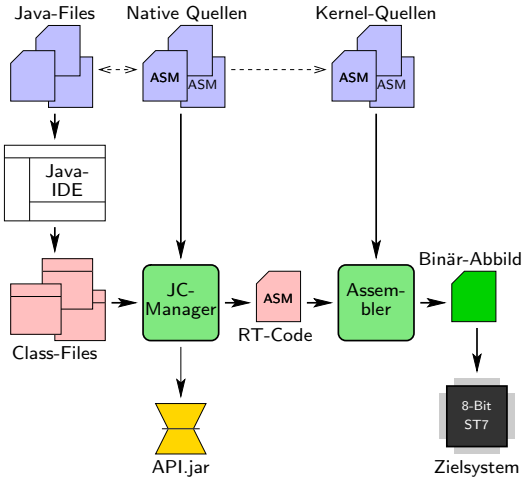


Bild 8.8: Entwicklungsablauf beim Zusammenstellen der Laufzeitumgebung und Assemblieren der VM

verlinker Namenskonflikte durch Ersetzung von Platzhaltern im nativen Code auf.

Block-Kommentare Der Code ist in viele Einzelroutinen aufgeteilt, von denen jede entsprechend der jeweiligen Sichtbarkeit auf alle Variablen zugreifen und diese verändern kann, hierfür findet keine Kontrolle mittels des Assemblers statt. Aus Platzgründen gibt es nur sehr wenige Variablen, die exklusiv von einzelnen Routinen verwendet werden, viele werden mehrfach genutzt. Damit es dabei nicht zu Konflikten mit aufrufenden Routinen kommt, verfügt jede Routine über einen ausführlichen Block-Kommentar, in dem vermerkt ist, welche Variablen in der Parameterliste stehen, welche Variablen verändert werden und welche weiteren Unterroutinen aufgerufen werden. Dies wird teilweise von eigenen Entwicklungswerkzeugen unterstützt und geprüft (siehe nächster Abschnitt).

Namenskonventionen Um die Unterscheidbarkeit der Symbole und die Lesbarkeit des Quelltextes zu verbessern, werden an der ungarischen Notation orientierte Namenskonventionen für alle Symbole verwendet. Sie setzen sich aus je einem Typ- und Modulpräfix, sowie dem eigentlichen Symbolnamen zusammen.

Konfigurationen Um dem Bedarf nach vielen Varianten der Zielsysteme zu begegnen, wurden globale Einstellungen in Konfigurationsdateien zusammengefasst. Varianten entstehen, wenn unterschiedliche Controller-Typen mit verschiedenen Zusammenstellungen der Peripheriekomponenten verwendet werden und wenn unterschiedliche externe Hardware eingesetzt wird. Die Konfigurationsdatei ist genauso wie die Zusammenstellung der Laufzeitumgebung durch den Vorverlinker ein Parameter des Build-Vorgangs einer spezifischen VM.

Trotz dieser Maßnahmen zur Fehlervermeidung und entsprechender Sorgfalt bei der Programmierung ist ein komplexes System in der Assemblersprache prinzipbedingt fehleranfälliger, als ein in einer Hochsprache formuliertes. Das ist ja auch ein Grund, Java auf kleinen eingebetteten Systemen zu etablieren. Schließlich wurden einige Maßnahmen zur Verifikation der virtuellen Maschine entwickelt.

8.3.3 Validierung

Die Fehlersuche auf eingebetteten Systemen mit nur wenigen Kommunikationsmöglichkeiten zum Entwickler, gestaltet sich schwierig, auch wenn Chip-Emulatoren mit Debugging-Schnittstellen zur Verfügung stehen. Die verfügbaren Werkzeuge sind nicht für komplexe Systeme mit eigener Speicherverwaltung geeignet, so dass zusätzliche Werkzeuge geschaffen werden mussten. Hierbei wurden drei Strategien verfolgt:

1. Fehlervermeidung durch Werkzeugunterstützte Code-Inspektion,
2. Analyse von Speicherauszügen laufender Debugging-Sitzungen,
3. High-Level Funktionsprüfung einzelner Komponenten der virtuellen Maschine (*Unit-Tests*).

Um den fehlenden Kontrollen der Assemblersprache bzgl. Code-, Speicher- und Variablenverwendung zu begegnen, wird Code-Inspektion eingesetzt. Hierbei werden alle in einer Routine verwendeten Ressourcen festgestellt und zur späteren Verwendung in den Block-Kommentar eingetragen. Um diesen Vorgang zu unterstützen, wurde die Arbeit [Müc04] angeregt, in der ein Framework für die integrierte Entwickungsumgebung Eclipse geschaffen wurde, das eine Analyse des Quelltextes durchführt. Ergebnis der Analyse sind verschiedene Tabellenansichten der verwendeten Variablen mit Zugriffsart, wobei auch aufgerufere Routinen und Makros berücksichtigt werden. Gerade die Analyse der nicht sichtbaren Quelltext-Elemente hat sich hierbei als sehr hilfreich erwiesen, Konflikte beim Zugriff auf geteilte Ressorucen aufzudecken.

Bei der Analyse von Fehlern bei der Implementierung zur Laufzeit der VM in einem Chip-Emulator ist die Kenntnis des vollständigen Zustands der virtuellen Maschine sehr hilfreich. Hierzu kann aus einem Speicherauszug und den Projektdateien, mit denen die VM erzeugt wurde, eine für einen menschlichen Betrachter lesbare Strukturansicht des Heaps generiert werden. Dabei hat sich gezeigt, dass nur in seltenen Fällen ein Fehler im Heap sofort nach der Entstehung auffällt. Ein fehlerhafter Code kann ja Daten an jeder beliebigen Adresse verändern, daher lässt der Ort des Fehlers im Heap selten Rückschlüsse auf den Verursacher zu. Zur Eingrenzung von Fehlern kann die virtuelle Maschine optional zusätzliche Überprüfungen durchführen, die über die in der Spezifikation definierten hinausgehen. Beispielsweise können bei jedem Zugriff die Handles im Heap auf Gültigkeit geprüft werden.

Um die Einhaltung der Spezifikation [LY00] zu gewährleisten, wurde ebenfalls in [Müc04] ein Java-Testmustergenerator entwickelt, der es erlaubt, Klassendateien mit allen Bytecodes und dedizierten Datenfeldern zu erzeugen. Auf diese Weise ist es möglich, im normalen Java-Programmieralltag mit gewöhnlichen Java-Compilern nur sehr unwahrscheinlich auftretende Situationen zu provozieren. Der Code-Generator erzeugt dabei eine komplette Test-Suite, die automatisch auf dem Zielsystem ablaufen kann und einen Report generiert. Dieser Test umfasst die Java-Schnittstelle der VM, nativer Code oder die Laufzeitbibliothek werden dadurch nicht abgedeckt.

8.3.4 Remote-Debugging

Auf dem ST7-Zielsystem existiert noch eine Erweiterung der Yogi2-VM. *Sun* hat eine Debugging-Schnittstelle für in der VM laufende Java-Anwendungen spezifiziert, die *Java Platform Debugger Architecture* (JPDA) [28]. Diese Architektur verwendet ein Client-Server-Prinzip, bei dem die Benutzeroberfläche (Frontend) außerhalb der JavaVM läuft und das Java Debug Interface (JDI) bereit stellt. Auf der JavaVM mit dem zu testenden Java-Programm (Backend) wird ein Java Virtual Machine Debug Interface (JVMDI) realisiert. Zwischen diesen beiden Komponenten findet eine Kommunikation über einen nicht festgelegten Kanal (üblicherweise TCP/IP) mit einem festgelegten Protokoll (Java Debugger Wire Protocol; JDWP) statt. Die beiden Teile des Debuggers können auch auf unterschiedlichen Systemen ausgeführt werden (*Remote-Debugging*), so dass sich die JPDA grundsätzlich für eingebettete Systeme eignet. Für das ST7-Zielsystem sind jedoch die Anforderungen zu hoch:

- Das Kommunikationsprotokoll ist zu umfangreich und kann nur in einer gekürzten Fassung auf dem Zielsystem implementiert werden. Ferner erfordert das Debugger-Backend auch Ressourcen zur Laufzeit, die der zu

prüfenden Anwendung schließlich fehlen. Auch hier muss auf einige Teile verzichtet werden.

- Aus Platzgründen wurden vom Vorverlinker die Debugging-Informationen und die Symbole aus den Klassendateien auf dem Zielsystem entfernt und stehen dort somit auch dem Debugger-Backend nicht zur Verfügung.

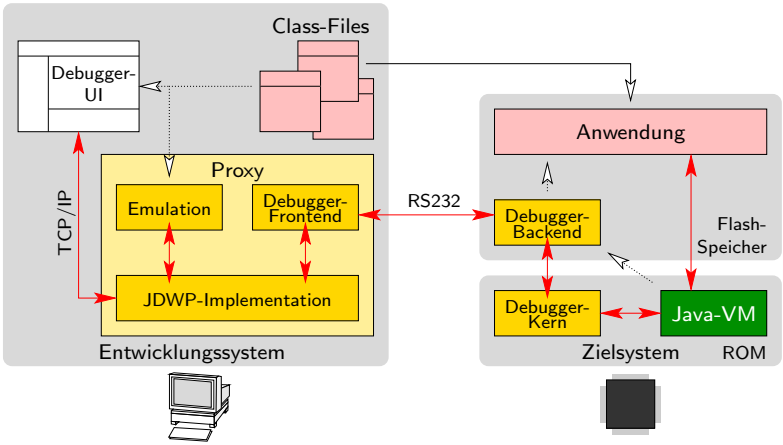


Bild 8.9: Remote-Debugging auf dem ST7-Zielsystem

Dennoch konnte hier ein Debugger durch nochmalige Aufteilung des Debugger-Backends in drei Teile realisiert werden (siehe Bild 8.9 und [Vel04]). Der größte Teil wurde vom Zielsystem auf das Entwicklungssystem verlagert und stellt die kompatible Schnittstelle zu einem beliebigen JDI-kompatiblen Frontend dar. Dieser *Debug-Proxy* hat zwei Aufgaben: Zunächst findet eine Protokollumsetzung statt, um die JavaVM zu entlasten und um den Kommunikationsoverhead zu minimieren. Ferner werden die Anfragen, die die VM nicht ausführen kann, bereits im Proxy bearbeitet. Dazu sind dem Proxy die Projektdatei und alle Klassendateien der auf dem Zielsystem laufenden Anwendung bekannt; da hier die Symbole und Debugging-Informationen (Zeilennummern, Variablennamen) noch vorhanden sind, können derartige Informationen auch ohne die VM beschafft werden. Ein Teil des Backends wird also emuliert. Anfragen, die den aktuellen Status des Programms betreffen (laufende Threads, Werte von Variablen) oder die Behandlung von Debugging-Ereignissen (Breakpoints), werden an die VM weitergeleitet. Dazu dient ein verkleinertes Frontend, mit dem die VM über ein vereinfachtes Protokoll kommuniziert, das an dem JDWP angelehnt ist.

Die übrigen zwei Teile befinden sich auf dem Zielsystem. Der Debugger-Kern (`DebuggerCore`) ist in den Kern der VM und in das Laufzeitsystem integriert, wofür eine entsprechende VM-Konfiguration verwendet wird. Der Debugger-Kern interagiert direkt mit der VM und stellt u. a. Funktionen zur Verfügung, um die Liste der Threads und Klasseninformationen zu erhalten, z. B. die interne Methodenrepräsentationstabelle zur Umwandlung von Adressen in Zeilennummern. Ferner können Objekte und die Rahmen laufender Methoden direkt als Speicherabbilder ausgelesen werden. Sie werden dann im Debug-Proxy zwischengespeichert und weiterverarbeitet, um die Werte von lokalen Variablen und Objektvariablen zu erhalten.

Tiefere Eingriffe in die VM sind bei der Umsetzung des Anhaltens einzelner Threads, von Einzelschritt-Ausführungen und von Breakpoints nötig. Das Anhalten von Threads ist zu einem beliebigen Zeitpunkt durch Ändern der Flags im jeweiligen `Thread`-Objekt einfach möglich. Für das automatisierte Anhalten ist jedoch eine Überprüfung des Java-Programmzählers (`JPC`) in der Hauptschleife des Bytecode-Interpreters nötig. Damit das ohne Leistungseinbußen geschieht, wenn keine Breakpoints gesetzt sind, wird der in Abschnitt 6.1 eingeführte Mechanismus der Bytecode-Interrupts genutzt. Wenn Debugging im Kern aktiviert ist, verfügt jeder Thread über zusätzliche Flags. Das Überprüfen der Flags fällt nur beim Threadwechsel an, ist also wesentlich weniger Performancekritisch als eine Überprüfung bei jedem Bytecode. Die Thread-Flags werden entweder direkt gesetzt (Einzelschritt) oder implizit beim Betreten von Methoden, in denen Breakpoints definiert sind (Trace). Beim Selektieren von Methodenrahmen fallen also weitere geringe Kosten an. Nur wenn der Trace-Modus aktiviert ist, wird nach jedem ausgeführten Bytecode ein Interrupt in den Debugger-Kern ausgelöst, wo die Adresse des Breakpoints mit dem `JPC` verglichen wird. Damit diese Überprüfung auch bei mehreren definierten Breakpoints effektiv ist, wird eine Hashtabelle verwendet, das gilt auch bei der Überprüfung der Methoden-Adressen beim Betreten der Rahmen.

Um den Speicherplatzbedarf auf dem Zielsystem zu verringern, wenn Debugging nicht erforderlich ist, wurden Teile des Backends in den wiederbeschreibbaren Anwendungsspeicher ausgelagert. Hier findet die Umsetzung des vereinfachten Kommunikationsprotokolls über die serielle Schnittstelle statt. Um den Debugger zu aktivieren, wird zusätzlich zur Anwendung ein weiteres Archiv geladen, das `Debugger` enthält. Es liegt in der Suchreihenfolge vorne, so dass zunächst das Debugger-Backend gestartet wird. Erst bei Verbindungsaufnahme durch das Frontend wird die zu prüfende Anwendung aus dem in der Suchreihenfolge nächstgelegenen Archiv gestartet.

Kapitel 9

Zwischenbilanz

9.1 Realisierungen und Anwendungen in der Praxis

Die hier realisierte Implementierung einer 8-Bit-JavaVM ist unter der Bezeichnung JCVM8 Grundlage für einige Produkte der Firma `domo:logic`. Sie werden zusammen mit der hier vorgestellten Programmierschnittstelle und einer integrierten Entwicklungsumgebung unter dem Namen *JControl* angeboten (siehe Bild 9.1):

JControl/Stamp Hierbei handelt es sich um eine kleine Einsteckplatine im DIL-Format. Einsatzgebiet sind eingebettete Steuerungen ohne wesentliche Benutzerinteraktion. Es gibt hiervon auch eine Variante mit einer CAN-Bus-Schnittstelle.

JControl/SmartDisplay Dieses Modul verfügt über ein grafisches Display mit 128×64 Pixeln und ist für Bedieneinheiten und zur Datenvisualisierung gedacht. Die Bedienung kann über eine externe Tastatur oder ein Touch-Panel erfolgen. Wie die Stamp ist das SmartDisplay für den Einsatz in vorhandenen Schaltungen gedacht.

JControl/PLUI Das Power-Line User-Interface ist ein Komplettgerät und die Kombination aus einem SmartDisplay und einem Power-Line-Modem (Konnex Bus Coupling Unit, BCU). Es kann in Gebäuden eingesetzt

Bild 9.1: Einige *JControl*-Produkte (Stamp, SmartDisplay, PLUI)

werden, die über Geräte mit einer passenden Power-Line-Anbindung verfügen. Auch Messungen der Kommunikation über das Stromnetz können vorgenommen werden.

Es entstand eine eigenständige Java-Familie. Teilweise sind die Produkte beim Endkunden im Einsatz (z. B. das PLUI bei der Firma Siemens zur Analyse von Power-Line-Kommunikation). Es werden die *JControl*-Module aber auch von privaten und gewerblichen Entwicklern nachgefragt. So findet sich das *JControl*/SmartDisplay unter dem Namen Java-Control-Unit (JCU10) im Katalog der Firma ELV, einem Endkunden-Distributor von elektronischen Komponenten und Geräten, wieder. Es hat sich dabei inzwischen eine kleine Gemeinschaft von Benutzern entwickelt, die ihre Erfahrungen und Probleme im Internet (u. a. auf [29]) austauschen. Dort ist es interessant zu beobachten, welch vielfältige Ideen auf dieser Plattform umgesetzt werden und was für neue Anwendungen sich zeigen. Diese Gemeinschaft macht das Projekt sogar für einen Markt von Sekundärliteratur interessant [Hau06].

9.2 Weiterführende Entwicklungen

Schließlich werden bei `domo:logic` in fortgesetzter Kooperation mit der Abteilung Entwurf integrierter Schaltungen weitere Produkte entwickelt, die auf den hier vorgestellten Technologien der Yogi2-VM basieren. Die JCVM32 ist eine Adaption unter Verwendung der Programmiersprache **C**. Varianten der JCVM32 können unter Windows und Linux auf dem PC und uClinux auf eingebetteten Systemen eingesetzt werden. Eingesetzt wird die JCVM32 auf der firmeneigenen Plattform Colibree, der Kombination einer Freescale[30] ColdfireCPU und eines Xilinx SpartanXL FPGAs, die mit 54 MHz getaktet werden. Diese Plattform verfügt über 2 MByte Speicher und profitiert somit weniger von den hier vorgestellen Methoden zur Einsparung von Speicherplatz. Auf Basis dieser Plattform wurden auch weitere Produkte entwickelt. Es entstanden zahlreiche Bibliotheken für grafische Bedienoberflächen, zur Videosignalverarbeitung und Kommunikation (CAN, Bluetooth). Ferner wurde eine Software-Zwischenschicht geschaffen, um eine Kompatibilität zur Java2 Micro Edition (J2ME) zu erlangen.

In Zukunft sollen auch weitere kleinere Mikrocontroller, ebenso preiswert wie der ST7, aber moderner und leistungsfähiger z. B. auf der ARM-Familie [30] basierend, eingesetzt werden. Auf der ST7-Familie wird lediglich noch Modellpflege betrieben, neue Eigenschaften, die über in Java implementierte neue Programmbibliotheken hinausgehen, sind unwahrscheinlich.

[30] Ehemals Motorola

9.3 Erfahrungen

Ziel dieser Arbeit war die Programmiersprache Java auch auf kleinen eingebetteten Systemen auf Mikrocontrollerbasis verfügbar zu machen. Dabei mussten die Einschränkungen dieser Plattform (Speicherplatz, Rechengeschwindigkeit) mit den Anforderungen von Java vereinbart werden. Es hat sich gezeigt, dass die Verwendung von Java mit dieser kleinen virtuellen Maschine die Programmentwicklung für Mikrocontroller gegenüber Assembler oder **C** enorm beschleunigen und bereichern kann [BKT03b]. Natürlich gibt es Einschränkungen bei der Arbeitsgeschwindigkeit, so dass dieses System für Anwendungen mit höheren Rechenanforderungen weniger gut geeignet ist, genügsame Anwendungen finden sich jedoch zuhauf, so dass die Vorteile überwiegen.

Die vorhandenen und bei `domo:logic` weiterentwickelten Entwicklungswerkzeuge unterstützen den Anwendungsprogrammierer. Etwas schwieriger gestaltet sich noch die Simulation der Anwendungen auf dem Entwicklungssystem ohne Verwendung des eingebetteten Systems selbst, das beim Remote-Debugging immer erforderlich ist. Die Verwendung realer Hardware beim Entwickeln von Anwendungen kann je nach Art der angeschlossenen Hardware auch eine Gefahr für die Hardware oder gar den Benutzer darstellen, so dass eine reine Simulation vorzuziehen ist. Es existiert ein einfacher Simulator, der die Anwendungen in einer Emulation der Laufzeitbibliothek ausführt. Angeschlossene Hardware wird mittels grafischer Komponenten nachgebiltet, die verschaltet werden können. Da hierbei die virtuelle Java-Maschine des Entwicklungssystems direkt verwendet wird, werden einige Fähigkeiten der realen eingebetteten VM nicht nachgebildet, z. B. die Echtzeitunterstützung.

Das eigentlich verfolgte Ziel, der weitgefächerte Einsatz von Java in der Home-Automation, wurde noch nicht erreicht. Das Problem ist die Anbindung an standardisierte Home-Automation-Busse, die nicht direkt von dieser VM geleistet werden kann. Hierfür kommen z. Zt. Systeme aus zwei Controllern zum Einsatz (PLUI). Sie sind allerdings noch zu kostspielig für den Einbau in Lichtschaltern oder Endgeräten. Der Einsatz von Controllern neuerer Generationen wird dieses Manko beseitigen. Schon heute existieren zahlreiche Kommunikationskomponenten in der Laufzeitbibliothek, die nur noch an die neue Hardware angepasst werden müssen.

Kapitel 10

Ausblick

Die Yogi2-VM hat das Ziel erreicht, mittels Java Anwendungen auf kleinen eingebetteten Systemen effizienter und sicherer entwickeln zu können. So einfach die Erstellung von Anwendungsprogrammen für und mit dieser JavaVM ist, so schwierig gestaltet sich die Erstellung, Weiterentwicklung und Wartung der VM selbst und von deren Zusatzkomponenten. Die Implementierung ist hochgradig zielsystemabhängig und liegt selbst nicht in Java vor, die bei Java vorhandenen Vorteile können dort also nicht genutzt werden. Oft existieren bei verschiedenen Zielsystemen komplett unterschiedliche Code-Basen, d. h. Erweiterungen oder Änderungen müssen getrennt eingepflegt werden. Das gilt insbesondere für den nativen Code, der feingranular mit der VM verwoben ist. Daraus ergibt sich bei der Realisierung auf unterschiedlichen Zielsystemen ein Konsistenzproblem der Programmierschnittstelle (siehe Bild 10.1). Ein zielsystemunabhängiges Modell der VM, das erst in einem letzten Schritt in eine zielsystemabhängige Implementierung umgewandelt wird, könnte diese Einschränkungen und Probleme vermeiden.

Ein weiterer Schwachpunkt ist, wie schon erwähnt, die nur teilweise mögliche Simulation von Anwendungen auf dem Entwicklungssystem ohne Verwendung des eingebetteten Systems selbst. Die vorhandene Simulationsschicht ist nicht in der Lage, die angeschlossene Hardware vollständig nachzubilden und verwendet eine andere virtuelle Java-Maschine. Eine vollständige Nachbildung kann nur gelingen, wenn ein zyklengenaues Modell der eingebetteten VM existiert. Erst dann kann eine Analyse des Verhaltens von Anwendungen durchgeführt und das System zu einer harten Echtzeitunterstützung ausgebaut werden. In diesem Kapitel sollen daher Überlegungen für einen modellbasierten Ansatz zur Beschreibung von virtuellen Java-Maschinen für kleine eingebettete Systeme durchgeführt werden.

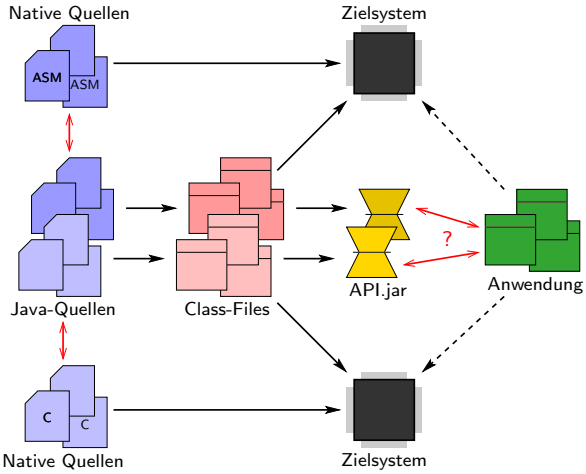


Bild 10.1: Nebenläufiger Entwicklungsfluss der Yogi2-Implementierung

10.1 Verwandte Arbeiten

Es existieren bereits einige Generatoren für virtuelle Maschinen, die wichtigsten sollen hier in Betracht gezogen werden. Im OpenVM-Projekt (OVM) [31] entsteht ein Framework zur Generierung anwendungsspezifischer VMs. Dabei wird eine Java-Anwendung als Grundlage verwendet, um eine spezielle VM zu erzeugen, die nur diese eine Anwendung ausführen kann. Es wird direkt zielsystemabhängiger Code erzeugt (Ahead-of-Time-Compiler). Ähnliche Projekte sind FLEX [32] und der GCJ der Gnu Compiler Collection [33]. Mit Erweiterungen wie jRate [34, CS02] ist der GCJ auch geeignet für Echtzeitverarbeitung.

Ein weiteres Projekt verfolgt einen etwas anderen Ansatz, die Jikes Research Virtual Machine (RVM) [35, A$^+$00, A$^+$05]. Es handelt sich um eine vollständig in Java implementierte virtuelle Maschine, die beim Klassenladen Java-Bytecode in nativen Code übersetzt und damit größtenteils auch sich selbst. Auf eingebetteten Systemen stößt die Jikes RVM jedoch an ihre Grenzen, denn ein Großteil der Technologie dieser VM liegt in den (zwar in Java implementierten, aber stark plattformabhängigen) Compilern.

Alle genannten Systeme bieten (teilweise mit Anpassungen) die automatisierte Generierung von virtuellen Java-Maschinen. Allerdings werden für die hier angestrebten Ziele einige Punkte nicht oder nur schlecht erfüllt:

- Der Speicherverbrauch ist zu hoch. Die vollständige Übersetzung in nativen Code verhindert die in Abschnitt 4.2.1 genannten Vorteile von Java-Bytecode bei der Einsparung von Speicherplatz. Eine Übersetzung zur Laufzeit sorgt gar für einen großen Bedarf an flüchtigem Speicherplatz zur Ablage des übersetzten Bytecodes.
- Die Konfigurierbarkeit ist sehr beschränkt. Gerätespezifische Varianten der VM und der Laufzeitbibliothek sind nicht vorgesehen. Ggf. wird nur eine Zielarchitektur oder -sprache unterstützt.
- Es besteht nur eine lose Koppelung mit dem Zielsystem. Um Hardwarekomponenten anzusteuern, müssen also zusätzliche native Bibliotheken implementiert werden, was also keinen Vorteil gegenüber dem Yogi2-Konzept bietet, die Code-Granularität dürfte dabei weniger fein sein.
- Eine Simulation von Anwendungen am Entwicklungssystem wird nicht unterstützt. Evtl. bestehen Möglichkeiten zur Analyse der Ausführungszeiten von Echtzeit-Aufgaben.

Jeder dieser Punkte für sich disqualifiziert die genannten Werkzeuge für den hier gewünschten Zweck eines zielsystemkonfigurierbaren Generators für kompakte JavaVMs.

10.2 Generische virtuelle Java-Maschinen für eingebettete Systeme

Ein neuer Ansatz soll modernere Softwaretechniken benutzen und offener, vielseitiger, fehlerfreier, wartbarer und schneller implementierbar werden. Wie das möglich wird, soll im Folgenden gezeigt werden. Eine ausführliche Beschreibung dieses Konzepts findet sich in [Böh06a].

10.2.1 Anforderungen an eine modellbasierte VM-Generierung

Unter Berücksichtigung der festgestellten Fähigkeiten und Einschränkungen bisheriger Implementierungen sollen zunächst die Anforderungen an den neuen Ansatz zusammengestellt werden:

Geschlossenes Framework Es enthält ein *Modell* der virtuellen Maschine, verwaltet die Varianten für unterschiedliche Konfigurationen und Zielsysteme und stellt auf dem Entwicklungssystem eine Simulationsumgebung bereit.

Durchgängige Implementierungssprache Die Integration aller Quelltexte in ein gemeinsames Modell vermeidet Konsistenzprobleme. Teil des Modells ist die vorhandene Laufzeitbibliothek, die die später auf dem System verfügbare *Programmierschnittstelle* (API) bildet. Der notwendige Anteil nativen Codes wird auf ein Mindestmaß verringert. Die einzigen nativen Komponenten sind der Bytecode-Interpreter, die Speicherverwaltung, der Thread-Dispatcher und ggf. noch eine schlanke Hard-/Software-Schnittstelle. Um die Bindung der nativen Komponenten an den Java-Code zu erhöhen, soll der native Code in den Java-Quelltext integriert werden.

Zielsysteme Die plattformunabhängige Beschreibung der virtuellen Maschine wird erst in einem letzten Schritt in eine zielsystemabhängige Form gebracht. Dabei werden die Klassendateien wie gehabt optimiert (vorverlinkt) und ggf. in ein spezielles zielsystemabhängiges Format umgewandelt. Der vorhandene native Code wird darin integriert. Es besteht eine *enge Koppelung* zum Zielsystem, Eigenheiten der CPU, des Speichers und der Hardwarekomponenten sind bekannt und werden genutzt. Die Übersetzung einer Beschreibung des nativen Codes erledigt ein Satz von *Code-Generator*en, deren APIs in Java verfügbar sind.

Datentypen Das Modell stellt einen Satz von Datentypen bereit, die Hardware-Konfigurationen zugeordnet werden können. Es soll sowohl spezielle Typen für hardwarenahe Datenstrukturen geben als auch für die interne Verwendung des Kerns der VM. Beispielsweise werden allgemeine Typen für Referenzen oder die in Java festgelegten primitiven Zahlenformate definiert, die im Modell durchgängig verwendet werden, auch im nativen Code. Das garantiert die *Typsicherheit* des Modells. Anschließend findet eine Abbildung dieser Standard-Typen auf zielsystemabhängige Varianten statt, z. B. mit reduzierter Bitzahl. Ferner sollen spezielle Datentypen verwendet werden, die *Speichermodelle* beispielsweise für den Zugriff auf Heap-Speicher, lokalen Speicher von Peripheriekomponenten oder externen Programm- und Grafikspeicher realisieren. Diese Speichermodelle können einen Großteil des sonst notwendigen nativen Codes ersetzen.

Varianten Basis des Systems soll der Baum des APIs sein, das auf den Zielsystemen realisiert werden soll, z. B. das *JControl*-API. Dieser Baum existiert nur einmal, so dass eine Änderung nur an einer einzigen Stelle erfolgen muss. In diesen Baum werden gleichermaßen Zielsystem-Varianten und Bibliotheks-Varianten eingepflegt. Dabei werden die spezialisierenden Varianten mittels Vererbung dem API-Baum hinzugefügt (Strategie-Entwurfsmuster, [GHJV95]). Damit diese Vererbung nicht auf die Zielsysteme gelangt, sollen die Varianten zur Übersetzungszeit in die API-Klassen integriert werden (statische oder *verdeckte Vererbung*). Die dafür

notwendige *Code-Migration* kann durch Manipulation der Java-Klassendateien erfolgen, wofür Werkzeuge existieren.

Verifikation Um die Verifikation der VM zu vereinfachen, soll das Modell *schichtenorientiert* aufgebaut sein. Ist einmal die Funktionsfähigkeit der VM und ihrer Komponenten auf der untersten Verhaltensebene erreicht und die Konformität mit der Spezifikation [LY00] nachgewiesen, so gilt dies auch für alle davon abhängigen speziellen Varianten. Es muss lediglich der erforderliche zusätzliche Code der entsprechenden Zwischenschichten und Code-Generatoren für die jeweiligen Zielsysteme getestet werden. Dabei handelt es sich um sehr begrenzte und überschaubare Module.

Simulation Das Modell ist in jeder Schicht auch auf dem Entwicklungssystem *ausführbar* angelegt. Einer bestimmten VM-Zusammenstellung kann eine auszuführende Anwendung zugeordnet werden. Für die Simulation einer Anwendung wird also nicht direkt eine andere virtuelle Maschine (des Entwicklungssystems) verwendet, sondern das Modell der eingebetteten VM. Es handelt sich um eine VM in der VM. Um eine Interaktion der Anwendung mit dem Entwicklungssystem zu ermöglichen, existieren außerhalb des ansonsten geschlossenen VM-Modells Simulationsmodule.

10.3 Das Modell der virtuellen Maschine

Im Folgenden wird das aus diesen Anforderungen resultierende Modell der VM konkret skizziert. Ausgangspunkt für das Modell ist die Programmierschnittstelle (API), die später vom Anwendungsprogrammierer wahrgenommen wird. Wie Bild 10.2 anhand eines kleinen Auszugs der Klassenhierarchie zeigt, wird diese Schnittstelle um weitere Komponenten des VM-Kerns und der Laufzeitumgebung ergänzt. Diese Kompontenten sind nicht bei der Anwendungsentwicklung sichtbar, wohl aber bei der VM-Entwicklung und -Zusammenstellung. Entsprechend einer Konfiguration können mehr oder weniger fein modellierte Komponenten für eine Ausführung auf dem Entwicklungssystem vorgesehen werden. Es ergeben sich dabei mehrere Modell-Ebenen:

Verhaltensmodell Es wird lediglich die durch die Laufzeitbibliothek und der Programmierschnittstelle vorgegebene Funktionalität nachgebildet, nicht aber Einflüsse durch spezielle Hardware-Varianten oder Zielsysteme. Hier manifestieren sich bereits mögliche Varianten für Komponenten von VM und API. Das Verhaltensmodell ist im Prinzip fähig, eine Anwendung auszuführen: Diese wird gegen das resultierende API übersetzt und als Startparameter der VM angegeben.

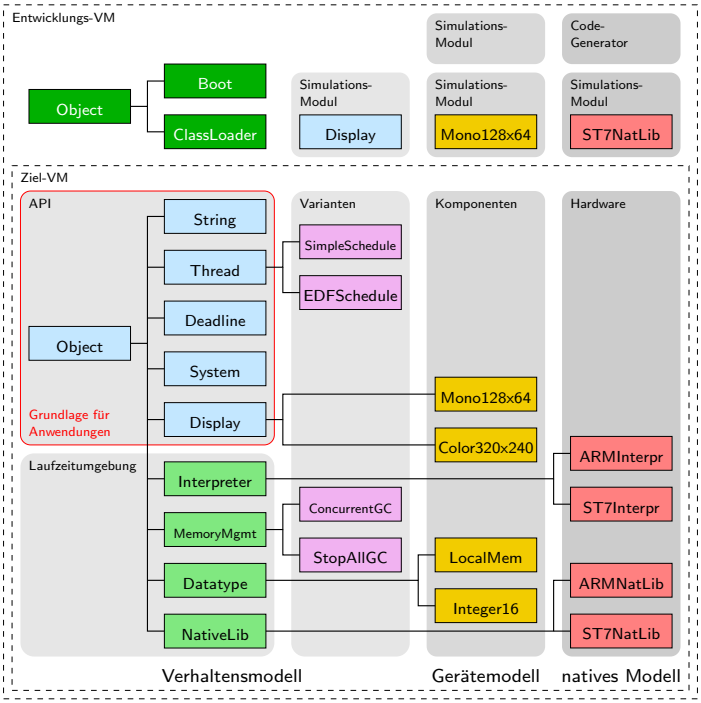


Bild 10.2: Klassenhierarchie des dreischichtigen Modells (Auszug)

Gerätemodell Dieses Modell bildet eine bestimmte Variante auf einem Zielsystem nach, das betrifft auch die exakte Speicherverwendung und die gesteuerte Peripherie, nicht aber das exakte zeitliche Verhalten.

Natives Modell Es wird das komplette Zielsystem inklusive des nativen Codes nachgebildet. Dabei kann das exakte zeitliche Verhalten einer Anwendung bestimmt und analysiert werden.

Außerhalb des VM-Modells existiert ein Satz von Simulationsmodulen und Code-Generatoren. Sie sind bestimmten Klassen des Modells zugeordnet. Die Code-Generatoren werden bei der Übersetzung des Modells aufgerufen. Simulationsmodule lösen bei der Ausführung des Modells bei Aktionen im Raum der Ziel-VM entsprechende Aktionen im Raum der Entwicklungs-VM aus. Sie bilden auch die unterschiedlichen Ebenen des Modells nach, so dass für bestimmte Varianten oder Zielsysteme auch passende Simulationsmodule existieren. Als

Beispiel ist ein grafisches Display zu nennen, das auf einer zielsystemunabhängigen Ebene keine festgelegte Auflösung und Farbtiefe hat und einfach die Aufrufe von Zeichenfunktionen des APIs in einem Fenster auf dem Entwicklungssystem darstellt. Auf einer zielsystemabhängigen Ebene wird hingegen direkt der Inhalt eines (nachgebildeten) Frame-Buffers dargestellt. Da auch für den nativen Code Simulationsmodule existieren, kann das Verhalten auf dem Zielsystem zeit- bzw. zyklengenau nachgebildet werden. Schließlich ist es nun auch möglich, zeitkritischen Code auf dem Zielsystem zu analysieren und die maximalen Ausführungszeiten (*Worst Case Execution Time*s; WCET) zu ermitteln.

10.4 Lösungsansätze und Techniken

Dieser Abschnitt behandelt einige Lösungsansätze einer konkreten Realisierung dieses Frameworks. Die hier vorgestellten Verfahren betreffen ausschließlich die Schichten des Verhaltens-, Geräte- und nativen Modells. Auf der Anwendungsschicht werden diese Mechanismen mit Ausnahme des Vorverlinkens zum Erzeugen eines vom Zielsystem verwendbaren Abbilds nicht mehr benötigt. Das Framework wird in zwei Durchgängen bearbeitet:

1. Das Modell wird ausgehend von einer Konfiguration in zusammengesetzte Klassendateien und Binärdateien für ein Zielsystem übersetzt. Dabei werden klassische Java-Compiler und spezielle Werkzeuge, die Teil des Frameworks selbst sind, verwendet.

2. Das übersetzte Modell wird entweder auf ein Zielsystem geladen und dort ausgeführt oder auf dem Entwicklungssystem benutzt, um eine Anwendung mit Hilfe der Simulationsmodule zu verifizieren.

Beide Vorgänge sollen durch Markierungen im Java-Quelltext gesteuert werden (*Annotations* wurden mit der Version 5 des JDK eingeführt) [36]. Annotations haben gegenüber externen Meta-Daten, die oft mit XML angegeben werden, den Vorteil, dass sie an der Stelle im Quelltext stehen, auf den sie sich beziehen. Ein Wechsel zwischen mehreren Dateien und Sprachen ist nicht mehr nötig, um eine Verknüpfung zwischen zwei Klassen oder Methoden herzustellen. Ferner haben Annotations gegenüber ähnlichen Mechanismen, die über die im Quelltext vorhandene Dokumentation funktionieren (Doclets), einen weiteren Vorteil: die Typsicherheit. Jede Annotation ist als Java-Klasse definiert und hat eine festgelegte Schnittstelle mit Elementen bestimmter Java-Typen. Moderne Java-Entwicklungsumgebungen wie z. B. Eclipse [37] unterstützen den Entwickler an dieser Stelle auch mit einem Inhaltsassistenten (Content Assist), so dass Fehler durch falsche Schreibweise oder Parameter auch bei Annotations von vornherein vermieden werden können.

Zur Verdeutlichung von Annotations in diesem Kontext steht folgendes Beispiel (mit einer angenommennen Klasse, die `Display` ableitet):

```
public @Migrate class MyDisplay extends Display {
  public void drawLine(int x1, int y1, int x2, int y2){
    [...]
  }

  public Position @Migrate(Pointer.class) getPointer(){
    [...]
  }
  [...]
}
```

Hier wurde die Klasse markiert, um in die Superklasse übernommen zu werden. Das betrifft hier nur die Methode `drawLine(..)`. Bei der Methode `getPointer()` gibt es eine Anweisung, sie in eine bestimmte andere Klasse zu migrieren. Mehr zur Code-Migration unten.

10.4.1 Annotation-Processing

Annotations haben noch eine interessante Eigenschaft: Sie werden vom Java-Compiler mit in die resultierenden Klassendateien aufgenommen, so dass sie von weiterverarbeitenden Werkzeugen benutzt werden können, ohne den Quelltext aufwändig analysieren und bearbeiten zu müssen. Dazu stellt *Sun* ein Werkzeug, das *Annotation Processing Tool* (APT), und eine Programmierschnittstelle (Mirror-API) zur Verfügung. APT wird dabei genauso wie der Java-Compiler aufgerufen und ersetzt ihn.[31] Zusätzlich werden aber noch selbst definierte Annotation-Prozessoren aufgerufen, die neue Quelltexte erzeugen oder fertig übersetzte Klassen bearbeiten können. Dabei können sogar eigene Fehlermeldungen erzeugt werden. Dieser Mechanismus wird benutzt, um entsprechend einer Konfiguration die Klassendateien des Modells bei dessen Übersetzung zu bearbeiten:

Code-Migration Ein Annotation-Prozessor bearbeitet die Klassen, die in einer angegebenen Konfiguration enthalten sind. Entsprechend markierte Klassen oder Methoden werden in die Superklasse oder eine angegebene Klasse verschoben. Dieser Vorgang ist rekursiv und die Herkunftsklassen werden anschließend aus der Konfiguration entfernt. Zur Verschiebung des Codes werden zwei Verfahren kombiniert: Zunächst bearbeitet APT die Annotations und generiert Listen der Klassendateien mit Migrationsaufträgen. Anschließend werden diese Klassendateien von Bytecode-

[31] *Sun* hat angekündigt, in zukünftigen Java-Versionen diese Funktionalität in den normalen Compiler zu integrieren.

Manipulationswerkzeugen bearbeitet, mögliche Kandidaten hierfür sind z. B. ASM [38] oder Javassist [39].

Datentypen Die Deklaration und Integration spezieller und hardwarenaher primitiver Datentypen soll ebenfalls mit Annotations gelöst werden. Hierzu werden einfach normale Variablendeklarationen mit entsprechenden Markierungen versehen:

```
public class MethodFrame {
  private @LocationPointer int jpc;
  private @LocationPointer int code;
  private Class methodClass;
  private int flags;
  private @StackIndex int previousSP;
  [...]
}

public class MyStruct {
  private static @LocalMem(address=0x42, width=2) int value;
  private static @LocalMem(width=1) int action;
  private static @LocalMem(len=8) byte[] buffer;
  [...]
}
```

In dem ersten Beispiel werden gewöhnliche Java-Variablen zu einer *Struktur* zusammengesetzt. Es wird ausgenutzt, dass beim Übersetzten eines Java-Programms die Reihenfolge der deklarierten Datenfelder erhalten bleibt.[32] Im zweiten Beispiel werden statische Variablen deklariert, die permanent zur Verfügung stehen. In diesem Fall handelt es sich um lokalen Speicher zur Ansteuerung einer Peripheriekomponente. Jedem Datentyp ist ein Stück nativer Code zugeordnet, um auf den jeweiligen Speicher zuzugreifen. Das gilt auch implizit für die gewöhnlichen Java-Datentypen, die zielsystemabhängig definiert werden können.

Simulationsmodule Die Zuordnung eines Simulationsmoduls zu einer Klasse oder einer Methode geschieht ebenfalls mit einer Markierung im Quelltext. Dessen Bearbeitung findet aber nicht nur zur Übersetzungszeit des Modells, sondern zur Laufzeit einer zu simulierenden Anwendung statt. Entscheidend beim Umgang mit den Simulationsmodulen ist, das Verhalten der durch das Modell spezifizierten virtuellen Java-Maschine nicht zu verändern. Daher werden die Simulationsmodule nicht von der modellierten virtuellen Maschine des Zielsystems ausgeführt, sondern von der des

[32] Es handelt sich dabei lediglich um einen Erfahrungswert. Die Einhaltung der Reihenfolge ist in der Sprachspezifikation nicht vorgeschrieben. Ggf. müssen also die verwendeten Java-Compiler auf dieses Verhalten geprüft und, falls dies nicht der Fall ist, die Reihenfolge durch eine Nachbearbeitung der Klassendateien korrigiert werden.

Entwicklungssystems. Die Übersetzung erfolgt ebenfalls nicht im Kontext des Frameworks, sondern in der der auf dem Entwicklungssystem vorhandenen Laufzeitbibliothek. Es besteht also Zugriff auf die (grafische) Bedienoberfläche des Entwicklungssystems.

Code-Generierung Die automatische Generierung effizienten und kompakten zielsystemspezifischen Codes ist ein Kernpunkt bei einer erfolgreichen Umsetzung des Modells auf eingebetteten Systemen. Hierbei sollen unterschiedliche Zielsysteme unterstützt werden, der zielsystemabhängige Code in den Java-Quelltext integriert werden und ein typsicherer Datenaustausch zwischen den beiden Welten stattfinden können. Eine Lösung hierfür bietet eine in Java verfügbare Programmierschnittstelle eines plattformunabhängigen Zwischencodes, deren Verwendung ein weiteres Beispiel zeigen soll:

```
import static NativeLib;
public class Utility {
  public static int impreciseAdd(int a, int b) {
    int c=a+b;
    synchronized(NativeLib.class) {
      load(temp,1);            // random range
      call("randomgenerator"); // uses temp
      add(c,c,temp);
    }
    return c;
  }
}
```

Die Klasse `NativeLib` stellt einen Satz von statischen Methoden und Variablen zur Verfügung, die normal mit Java verwendet werden können. Daraus ergibt sich zunächst ausführbarer Java-Code, der bei einer Simulation verwendet werden kann. Diesen *nativen Primitiven* können nun zielsystemabhängige Code-Generatoren zugeordnet werden, die bei der Erzeugung des Zielsystem-Abbilds aufgerufen werden und entsprechende native Code-Sequenzen erzeugen.

Dieses Framework befindet sich z. Zt. unter dem Arbeitstitel Yogi3 in der Entwicklung. Zunächst wird ein Prototyp erstellt, der das Zielsystem des Yogi2-Projekts (ST7) bei Erhaltung des *JControl*-APIs unterstützt. Das ermöglicht nicht nur einen direkten Vergleich der beiden Systeme sondern auch den Übergang vorhandener Produkte zur neuen VM-Technologie. Anschließend kann die Unterstützung modernerer Zielsysteme, die sich besser für Java eignen (z. B. ARM), prinzipbedingt leicht integriert werden.

Kapitel 11

Zusammenfassung

In dieser Arbeit wurde mit Yogi2 eine virtuelle Java-Maschine auf einem 8-Bit-Mikrocontroller realisiert und damit gezeigt, dass die Programmiersprache Java auch unter den dort herrschenden Randbedingungen (Speicherplatz, Rechengeschwindigkeit) praktikabel einsetzbar ist und den Entwicklungsprozess komplexer Anwendungen positiv beeinflusst.

Die Motivation zu dieser Arbeit bestand darin, den Entwicklungsaufwand für eingebettete Systeme und insbesondere für kleine 8-Bit-Systeme zu verringern. Dort spielen wegen hoher Stückzahlen die Systemkosten eine entscheidende Rolle. Das betrifft im zunehmenden Maße die Softwareentwicklung: Neue und komplexe Einsatzgebiete wie z. B. die Home-Automation erfordern aufwändige Software-Systeme. Bei größeren Zielsystemen konnte die Softwareentwicklungszeit mit modernen objektorientierten Hochsprachen, wie z. B. Java, um ein erhebliches Maß gesenkt werden. Daher entstand der Wunsch, diese Vorteile auch auf kleinen eingebetteten Systemen zu nutzen.

Die virtuelle Java-Maschine übernimmt auf eingebetteten Systemen die Aufgabe eines universellen Betriebssystems und stellt eine standardisierte Programmierplattform zur Verfügung. Die Abhängigkeit der Anwendungen vom Zielsystem wird verrringert oder gar abgeschafft. Die teilweise im Rahmen dieser Arbeit entstandene und auf die Einsatzgebiete Steuerung, Benutzerinteraktion und Kommunikation zugeschnittene Programmierschnittstelle unterstützt den Entwickler mit vorgefertigten Komponenten bei der Anwendungserstellung.

Es haben sich aber noch weitere, teilweise unerwartete Vorteile gezeigt: Auf einigen Zielsystemen bietet der Java-Bytecode eine kompaktere Repräsentation von Algorithmen als bei Verwendung des nativen Befehlssatzes. Komplexe Anwendungen werden so auf kleinen Systemen erst ermöglicht. Als wichtig hat sich auch die enge Koppelung nativen Codes an die virtuelle Maschine herausgestellt. Nativer Code ist zur Ansteuerung der peripheren Hardware, zur Geschwindigkeitsoptimierung und für den internen Betrieb der virtuellen Maschine

notwendig. Ein auf Bytecodeebene funktionierender schneller Umschaltmechanismus erlaubt einen häufigen Wechsel zwischen beiden Ebenen und ermöglicht es, die beiderseitigen Vorteile zu kombinieren.

Ein weiteres Augenmerk wurde in dieser Arbeit auf die automatische Speicherbereinigung (Garbage-Collection) gelegt. Realisiert wurde eine nebenläufige Mark-Compact Speicherbereinigung, die durch Verwendung von indirekten Blockreferenzen (Handles) und atomaren Code-Sequenzen stark vereinfacht werden konnte. Dabei hat sich gezeigt, dass ineffiziente Algorithmen zum Markieren belegter Blöcke und vor allem bei deren Kompaktierung eine Auswirkung auf die Gesamtperformance der virtuellen Maschine haben. Das trifft selbst bei sehr kleinen Heaps mit einer Größe von wenigen KByte zu, auch wenn die Speicherbereinigung unter der Obhut eines Echtzeit-Schedulers aufgerufen wird.

Unbedingte Voraussetzung für die Abwicklung der eingebetteten Steuerungs- und gleichzeitigen Kommunikations- und Bedienungsaufgaben ist eine genaue Beeinflussbarkeit des zeitlichen Ablaufs einer Anwendung. Dazu wurde ein zeitschrankenbasierter dynamischer Echtzeit-Scheduler in die virtuelle Maschine integriert, welcher nach Erfüllung der zeitlichen Vorgaben insbesondere eine faire Zuordnung der Rechenzeit auf die Aufgaben (Tasks) ermöglicht, auch wenn sie nicht kooperativ programmiert sind. Um die Erstellung von Aufgaben mit Zeitvorgaben zu erleichtern, wurde in dieser Arbeit ein neues Konzept vorgestellt, Zeitschranken mit Java-Codesequenzen zu verknüpfen: die Synchronisierung auf ein `Deadline`-Objekt. Mittels geeigneter Entwurfsmuster lassen sich mit dieser sehr schlanken Programmierschnittstelle einmalige, periodische, sporadische und verschachtelte Echtzeitaufgaben formulieren.

Im Rahmen dieser Arbeit entstand eine Referenzimplementierung einer virtuellen Java-Maschine auf dem ST7-Mikrocontroller, einem typischen 8-Bit-System. Diese Realisierung ist nicht nur eine Forschungsplattform, sondern wird auch von Endkunden eingesetzt.

Ziel der Überlegungen im Ausblick dieser Arbeit ist, die Vorteile von Java nicht nur für die Erstellung von Anwendungen auf *einem* Zielsystem zu nutzen, sondern auch bei der Erstellung der virtuellen Java-Maschinen für *viele* Zielsysteme. Der Kern der JavaVM und die Laufzeitumgebung werden dabei komplett in Java formuliert und spezifiziert. Ein mehrschichtiges Modell enthält auf der einen Seite eine plattformunabhängige Beschreibung und auf der anderen Seite die plattformabhängigen Realisierungen für bestimmte Zielsysteme. Da die zielsystemabhängigen Modellebenen von den verhaltensbeschreibenden abhängen, wird ein nebenläufiger Entwicklungsfluss für verschiedene Zielsysteme vermieden. Änderungen am Verhalten der VM wirken sich auf allen realisierten Zielsystemen aus. Die Verwendung derselben Programmiersprache *für* die

Zielsysteme und *auf* den Zielsystemen erhöht deren Koppelung, Programmcode kann gemeinsam verwendet werden. Zielsystemspezifischer Code wird mit im Modell integrierten Code-Generatoren aus plattformunabhängigen Beschreibungen erzeugt, die in den Java-Quelltext integriert werden können.

Das Konzept verspricht eine höhere Flexibilität. Neue Verfahren und Datenstrukturen lassen sich schneller umsetzen. Für jede Zielplattform ist lediglich ein Speichermodell, ein Code-Generator und ggf. die Ansteuerung von Peripheriekomponenten zu realisieren. Wenn ein exaktes Verhalten eines Zielsystems auf dem Entwicklungssystem nachgebildet werden soll, werden dedizierte Simulationsmodule ergänzt. Das Framework und daraus gewonnene Java-VMs versprechen auch die Verwendung von Software-Entwicklungswerkzeugen der nächsten Generation auf kleinen eingebetteten Systemen. So scheint diese Plattform mit entsprechenden Programmierschnittstellen für einen domänenspezifischen Software-Entwurf [LKT04] gut geeignet zu sein.

Literaturverzeichnis

[A+00] Alpern, Bowen [u. a.]: The Jalapeño virtual machine. In: *IBM Systems Journal* 39 (2000), Februar, Nr. 1, S. 211–238. – `http://www.research.ibm.com/journal/sj/391/alpern.pdf`

[A+05] Alpern, Bowen [u. a.]: The Jikes Research Virtual Machine project: Building an open-source research community. In: *IBM Systems Journal* 44 (2005), Nr. 2, S. 399–417. – `http://www.research.ibm.com/journal/sj/442/alpern.pdf`

[BCC+03] Bollella, Greg ; Canham, Tim ; Carson, Vanessa ; Champlin, Virgil ; Dvorak, Daniel ; Giovannoni, Brian ; Indictor, Mark ; Meyer, Kenny ; Murray, Alex ; Reinholtz, Kirk: Programming with non-heap memory in the real time specification for Java. In: *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. Anaheim, CA, USA : ACM Press, 2003. – `http://hdcp.org/Publications/GG-OOPSLA.pdf`. – ISBN 1–58113–751–6, S. 361–369

[Bec99] Beck, Kent: *Extreme Programming Explained: Embrace Change*. First Edition. Addison Wesley, 1999. – ISBN 0–201–61641–6

[BG04] Böhme, Helge ; Golze, Ulrich: Lightweight Firm Real-Time Extensions for Low Memory Profile Java Systems. In: *OTM 2004 Workshops: 2nd Java Technologies for Real-Time and Embedded Systems (JTRES)*. Ayia Napa, Zypern : Springer-Verlag, Oktober 2004 (Lecture Notes in Computer Science 3292). – `http://www.springerlink.com/link.asp?id=9kd61c043evjgvmt`. – ISBN 3–540–23664–3, S. 303–314

[BGJS00] Bracha, Gilad ; Gosling, James ; Joy, Bill ; Steele, Guy: *The Java™ Language Specification*. Second Edition. Addison-Wesley, 2000. – `http://java.sun.com/docs/books/jls`. – ISBN 0–201–31008–2

[BKMS98] Bacon, David F. ; Konuru, Ravi B. ; Murthy, Chet ; Serrano, Mauricio J.: Thin Locks: Featherweight Synchronization for Java. In: *SIGPLAN Conference on Programming Language Design and Implementation*, 1998. – `citeseer.ist.psu.edu/bacon98thin.html`, S. 258–268

[BKT03b] Böhme, Helge ; Klingauf, Wolfgang ; Telkamp, Gerrit: *JControl* – Rapid Prototyping und Design Reuse mit Java. In: *11. E.I.S.-Workshop*. Erlangen : VDE-Verlag, April 2003 (GMM-Fachbericht 40). – ISBN 3–8007–2760–9, S. 79–84

[BT01a] Böhme, Helge ; Telkamp, Gerrit: Embedded Control mit Java. In: *Embedded Intelligence*. Nürnberg : Design & Elektronik, Februar 2001, S. 43–52

[BT02] Böhme, Helge ; Telkamp, Gerrit: *JControl* – ein speichereffizienter Java-Ansatz. In: *Embedded Intelligence*. Nürnberg : Design & Elektronik, Februar 2002, S. 149–158

[BTG98] Böhme, Helge ; Telkamp, Gerrit ; Golze, Ulrich: Eine Java-VM für eingebettete 8-Bit-Systeme. In: *GI/ITG-Workshop: Java und eingebettete Systeme*. Karlsruhe : Forschungszentrum Informatik, September 1998 (FZI-Bericht 4-13-9/98), S. 97–117

[Böh98] Böhme, Helge: *Konzeption und Implementierung einer virtuellen Java-Maschine auf einem 8-Bit-Mikrocontroller für Steuerungsaufgaben in der Home-Automation*, Abteilung E.I.S., Technische Universität Braunschweig, Diplomarbeit, 1998

[Böh01] Böhme, Helge: Konzeption und Implementierung einer neuen virtuellen Java-Maschine auf dem 8-Bit-Mikrocontroller ST7 für Steuerungsaufgaben in der Home-Automation / Abteilung E.I.S., Technische Universität Braunschweig. 2001. – Forschungsbericht

[Böh02] Böhme, Helge: Home-Automation Praktikum im Sommersemester 2002 (*JControl*) – Aufgabenstellung / Abteilung E.I.S., Technische Universität Braunschweig. 2002. – Forschungsbericht

[Böh06a] Böhme, Helge: Generische virtuelle Java-Maschinen für eingebettete Systeme / Abteilung E.I.S., Technische Universität Braunschweig. 2006. – Forschungsbericht

[Böh06c] Böhme, Helge: Die Yogi2-VM und das *JControl*-API, Quelltexte und Dokumentation / Abteilung E.I.S., Technische Universität Braunschweig. 2006. – Forschungsbericht

[C+00] Czajkowski, Grzegorz [u. a.]: Application isolation in the Java Virtual Machine. In: *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA : ACM Press, 2000. – `http://portal.acm.org/citation.cfm?id=353195`. – ISBN 1–58113–200–X, S. 354–366

[CC03] Corsaro, Angelo ; Cytron, Ron K.: Efficient memory-reference checks for real-time java. In: *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*. San Diego, California, USA : ACM Press, 2003. – `http://www.cs.wustl.edu/~corsaro/papers/LCTES03.pdf`. – ISBN 1–58113–647–1, S. 51–58

[CS02] Corsaro, Angelo ; Schmidt, Douglas C.: The Design an Performance of the jRate Real-time Java Implementation. In: *International Symposium on Distributed Objects and Applications*, 2002. – `http://www.cs.wustl.edu/~corsaro/papers/DOA02.pdf`

[CSCM00] Clausen, Lars R. ; Schultz, Ulrik P. ; Consel, Charles ; Muller, Gilles: Java bytecode compression for low-end embedded systems. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22 (2000), Nr. 3, S. 471–489. – `http://www.mip.sdu.dk/~ups/papers/toplas00.pdf`. – ISSN 0164–0925

[DKL+00] Domani, Tamar ; Kolodner, Elliot K. ; Lewis, Ethan ; Salant, Eliot E. ; Barabash, Katherine ; Lahan, Itai ; Levanoni, Yossi ; Petrank, Erez ; Yanover, Igor: Implementing an On-the-Fly Garbage Collector for Java. In: *ACM SIGPLAN Symposium on Memory Management*, 2000. – `http://www.cs.technion.ac.il/~erez/Papers/cgc9.pdf`, S. 155–166

[GAKS05] Gruian, Flavius ; Andersson, Per ; Kuchcinski, Krzysztof ; Schoeberl, Martin: Automatic Generation of Application-Specific Systems Based on a Micro-programmed Java Core. In: *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*. Santa Fe, New Mexico : ACM Press, March 2005. –

http://www.jopdesign.com/doc/sac05.pdf. – ISBN 1–58113–964–0, S. 879–884

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995. – ISBN 0–201–63442–2

[GS05] Gruian, Flavius ; Salcic, Zoran: Designing a Concurrent Hardware Garbage Collector for Small Embedded Systems. In: *Advances in Computer Systems Architecture: 10th Asia-Pacific Conference, ACSAC 2005*, Springer-Verlag, October 2005 (Lecture Notes in Computer Science 3740). – http://www.springerlink.com/link.asp?id=3u711u446w1m7r04. – ISBN 3–540–29643–3, S. 281–294

[Hau06] Hausherr, Bruno: *Messen, Steuern und Regeln mit der Java Control Unit*. Franzis Verlag, 2006. – ISBN 3–7723–5357–6

[HP94] Hennessy, John L. ; Patterson, David A.: *Rechnerarchitektur, Analyse, Entwurf, Implementierung, Bewertung*. Vieweg, 1994. – ISBN 3–528–05173–6

[Kee99] Keeling, N.J.: Missed it! – How Priority Inversion messes up real-time performance and how the Priority Ceiling Protocol puts it right. In: *Real-Time Magazine* (1999), Nr. 99-4, S. 46–49. – http://www-md.e-technik.uni-rostock.de/ma/gol/bsys/pdf/nrt.pdf

[KWK02] Kwon, Jagun ; Wellings, Andy ; King, Steve: Ravenscar-Java: a high integrity profile for real-time Java. In: *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, ACM Press, 2002. – http://www.cs.york.ac.uk/ftpdir/reports/YCS-2002-342.pdf. – ISBN 1–58113–599–8, S. 131–140

[LKT04] Luoma, Janne ; Kelly, Steven ; Tolvanen, Juha-Pekka: Defining Domain-Specific Modeling Languages: Collected Experiences. In: *The 4th OOPSLA Workshop on Domain-Specific Modeling*, 2004. – http://www.dsmforum.org/events/DSM04/luoma.pdf

[Lor97] Lorenzen, Klaus F.: *Das Literaturverzeichnis in wissenschaftlichen Arbeiten*. FH Hamburg, Januar 1997. – http://www.bui.haw-hamburg.de/pers/klaus.lorenzen/ASP/litverz.pdf

[LY00] Lindholm, Tim ; Yellin, Frank: *The Java™ Virtual Machine Specification*. Second Edition. Addison-Wesley, 2000. – `http://java.sun.com/docs/books/vmspec`. – ISBN 0–201–43294–3

[Müc04] Mücke, Stefan: *Java in der Home-Automation: Eine Test- und Analysesoftware für die 8-Bit-JControl-VM*, Abteilung E.I.S., Technische Universität Braunschweig, Diplomarbeit, 2004

[Nil98] Nilsen, Kelvin: Adding real-time capabilities to Java. In: *Communications of the ACM* 41 (1998), Nr. 6, S. 49–56. – `http://www.acm.org/pubs/citations/journals/cacm/1998-41-6/p49-nilsen/`

[Nil03] Nilsen, Kelvin: Doing Firm Real-Time with J2SE APIs. In: *OTM 2003 Workshops: Java Technologies for Real-Time and Embedded Systems (JTRES)*, Springer-Verlag, Oktober 2003 (Lecture Notes in Computer Science 2889). – `http://www.springerlink.com/link.asp?id=yd13r3u3wne9ewfg`. – ISBN 3–540–20494–6, S. 371–384

[Pil02] Pilz, Markus: Earliest Deadline First Scheduling for Real-Time Java. In: *Embedded Systems Conference Europe*, 2002

[SBCK03] Schultz, Ulrik P. ; Burgaard, Kim ; Christensen, Flemming G. ; Knudsen, Jørgen L.: Compiling java for low-end embedded systems. In: *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*. San Diego, California, USA : ACM Press, 2003. – `http://www.mip.sdu.dk/~ups/papers/lctes03.pdf`. – ISBN 1–58113–647–1, S. 42–50

[Sch05] Schöberl, Martin: *JOP: A Java Optimized Processor for Embedded Real-Time Systems*, Technische Universität Wien, Dissertation, Jänner 2005. – `http://www.jopdesign.com/thesis/index.jsp`

[Sch06] Schoeberl, Martin: A Time Predictable Java Processor. In: *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006)*. Munich, Germany, March 2006. – `http://www.jopdesign.com/doc/jop_wcet.pdf`. – ISBN 3–9810801–0–6, S. 800–805

[Sie02] Siebert, Fridtjof: *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. aicas GmbH, Karlsruhe, 2002. – ISBN 3–8311–3893–1

[Tan01] Tanenbaum, Andrew S.: *Modern Operating Systems*. Second Edition. Prentice Hall, 2001. – ISBN 0–13–031358–0

[Top02] Topley, Kim: *J2ME in a Nutshell*. O'Reilly, 2002. – ISBN 0–596–00253–X

[Vel04] Velitchkov, Kalin: *Implementierung eines Debugger-Backends für die 8-Bit virtuelle Java Maschine JControl*, Abteilung E.I.S., Technische Universität Braunschweig, Studienarbeit, 2004

[Wil92] Wilson, Paul R.: Uniprocessor Garbage-Collection Techniques. In: *International Workshop on Memory Management*. St. Malo : Springer-Verlag, September 1992 (Lecture Notes in Computer Science 637). – `http://www.inf.fu-berlin.de/lehre/WS97/GC/papers/GCSurvey.ps.gz`, S. 1–42

Verzeichnis der Internetquellen

[1] *Konnex Association.* http://www.konnex.org/

[2] *Echelon Corporation.* http://www.echelon.com

[3] *Java Technology: The early years.* http://java.sun.com/features/1998/05/birthday.html

[4] *Sun Microsystems*: *Java™ Technology Products & APIs.* – http://java.sun.com/products/

[5] *Sun Microsystems*: *The Java™ HotSpot Virtual Machine White Paper.* – http://java.sun.com/products/hotspot/

[6] *Java Virtual Machines, Java Development Kits and Java Runtime Environments.* http://java-virtual-machine.net/other.html

[7] *Sun Microsystems*: *Java™ 2 Platform, Micro Edition (J2ME).* – http://java.sun.com/j2me/

[8] esmertec: *JBed.* – http://www.esmertec.com

[9] aicas: *JamaicaVM Realtime Java Technology.* – http://www.aicas.com/sites/jamaica.html

[10] *Sun Microelectronics*: *picoJava Microprocessor Cores.* – http://www.sun.com/microelectronics/picoJava/

[11] aJile Systems: *Products.* – http://www.ajile.com

[12] Schöberl, Martin. *JOP – Java Optimized Processor.* http://www.jopdesign.com

[13] *Sun Microsystems*: *Java Card Technology.* – http://java.sun.com/products/javacard/

[14] Maxim/Dallas Semiconductors: *TINI – Tiny InterNet Interface.* – http://www.maxim-ic.com/TINIplatform.cfm

[15] *simpleRTJ.* http://www.rtjcom.com

[16] Solorzano, Jose H. *TinyVM (RCX Java).* `http://tinyvm.sourceforge.net`

[17] *leJOS.* `http://lejos.sourceforge.net`

[18] Systronix: *JCX – Java Control System.* – `http://www.jcx.systronix.com`

[19] *muvium (PIC Java VM).* `http://www.muvium.com`

[20] *Yogi Tea – Green tea, organic herbal tea, black tea and Kundalini yoga.* `http://www.yogitea.com/`

[21] *Sun Microsystems*: *Tuning Garbage Collection with the 1.4.2 Java™ Virtual Machine.* – `http://java.sun.com/docs/hotspot/gc1.4.2/`

[22] *Caffeine Mark 3.0.* `http://www.benchmarkhq.ru/cm30/`

[23] *SPEC JVM98.* `http://www.spec.org/osg/jvm98/`

[24] Dibble, Peter ; Bollella, Greg ; Brosgol, Ben ; Belliardi, Rudy ; Wellings, Andy [u. a.]: *The Real-Time Specification for Java.* The Real-Time for Java Expert Group, Januar 2002. – `http://www.rtj.org`

[25] *JControl*: *JControl Platform API Specification, JCVM8 Edition.* – `http://www.jcontrol.org/docs/api/`

[26] ST Microelectronics: *ST7 Family Programming Manual.* – `http://www.eu.st.com/stonline/books/pdf/docs/4020.pdf`

[27] ST Microelectronics: *ST72311R Data Sheet.* – `http://www.eu.st.com/stonline/books/pdf/docs/6810.pdf`

[28] *Sun Microsystems*: *The Java™ Platform Debugger Architecture (JPDA).* – `http://java.sun.com/j2se/1.5.0/docs/guide/jpda/`

[29] *JControl*: *JControl-Homepage.* – `http://www.jcontrol.org`

[30] ARM: *Core Families.* – `http://www.arm.com/products/CPUs/families.html`

[31] Lea, Doug [u. a.]. *OpenVM.* `http://www.ovmj.org/`

[32] Rinard, Martin [u. a.]. *Flex Compiler Infrastructure.* `http://www.flex-compiler.lcs.mit.edu`

[33] Free Software Foundation: *GCJ: The GNU Compiler for the Java™ Programming Language.* – `http://gcc.gnu.org/java/`

[34] Corsaro, Angelo ; Deters, Morgan. *jRate: A Real-Time Java ahead-of-time compiler*. http://jrate.sourceforge.net

[35] Hind, Michael [u. a.]. *Jikes Research Virtual Machine (RVM)*. http://jikesrvm.sourceforge.net

[36] *Sun Microsystems*: *Annotations*. – http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

[37] *The Eclipse Workbench*. – http://www.eclipse.org

[38] Bruneton, Eric ; Kuleshov, Eugene ; Loskutov, Andrei [u. a.]. *ASM*. http://asm.objectweb.org

[39] Chiba, Shigeru [u. a.]. *Javassist*. http://www.csg.is.titech.ac.jp/~chiba/javassist/

[40] Czajkowski, Grzegorz [u. a.]: *JSR 121: Application Isolation API Specification. Sun Microsystems*. – http://www.jcp.org/en/jsr/detail?id=121

Index

Anhänge

Anhang A

Datenblatt der Yogi2-VM

Bei der Yogi2-VM handelt es sich um eine um die Realisierung einer virtuellen Maschine auf dem ST7-Mikrocontroller, die auf Technologien und Spezifikationen der Programmiersprache Java^{TM} der Firma *Sun Microsystems* [LY00] basiert. Mit ihr wird es möglich, vorhandene Java-Entwicklungswerkzeuge (Compiler, integrierte Entwicklungsumgebungen) für die objektorientierte Anwendungsentwicklung auf kleinen und preiswerten eingebetteten Systemen zu verwenden. Den grundsätzlichen Aufbau zeigt Bild A.1.

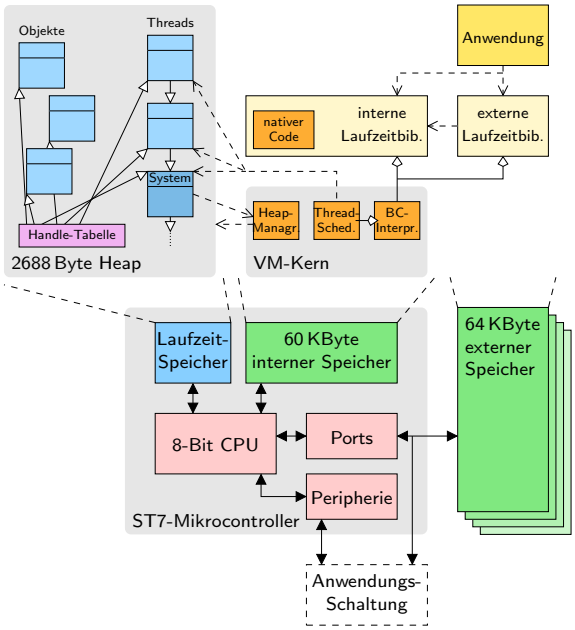


Bild A.1: Komponenten der Yogi2-VM

Komponenten

- Max. 2688 Bytes On-Chip Heap-Memory (Java-Laufzeitobjektspeicher).
- 60 KBytes interner (On-Chip) Programmspeicher, er enthält die JavaVM und Teile der Laufzeitbibliothek (fest einprogrammiert).
- Anschlussmöglichkeit für externen wiederbeschreibbaren Flash-Speicher der Größen 64, 128 und 256 KByte (z. B. die AT29C-Serie), welcher in Bänke zu je 64 KByte eingeteilt wird; die aktive Bank enthält Archive mit weiteren Teilen der Laufzeitbibliothek und die Anwendung.
- Ein kompakter Mikrokernel fasst Bytecode-Interpreter (Execution-Engine), Speicherverwaltung und echtzeitfähigen Thread-Scheduler zusammen.
- Die Laufzeitumgebung enthält die Klasseninitialisierung und die Speicherbereinigung (Garbage-Collection) in einem System-Thread.
- Eine interne Laufzeitbibliothek enthält die fundamentale Unterstützung der Java Programmiersprache sowie Klassen zur Ansteuerung von Peripheriekomponeten (z. B. das *JControl*-API).
- Es kommt 16-Bit Festkomma-Arithmetik zur Anwendung, die Java-Datentypen `byte`, `short`, `char` und `int` (16 Bit) werden unterstützt.

Bytecode-Interpreter

Der Bytecode-Interpreter führt Java-Bytecode im internen oder externen Festwertspeicher aus. Der Durchsatz liegt bei üblichem Betrieb im Bereich von 3300 bis 35000 Bytecodes/Sekunde, maximal sind ca. 170000 Bytecodes/Sekunde möglich, der Durchsatz ist bei der Ausführung von Code im externen Programmspeicher geringer als im internen. Im internen Programmspeicher kann nativer Code blockweise bezogen auf Bytecode-Grenzen eingefügt werden, aufgerufen wird er mit dem neuen Bytecode `callnative`. Für den Zugriff auf Java-Laufzeitdaten vom nativen Code existieren einfache und schnelle Code-Sequenzen, die über Makros verfügbar sind. Nativer Code kann beliebig mit Java-Bytecode kombiniert werden.

Speicherverwaltung

Die Speicherverwaltung organisiert den Heap. Die Adressierung der Speicherblöcke (sie enthalten Objekte) erfolgt indirekt über eine Handletabelle. Die

Verschiebung der Blöcke zur Laufzeit wird ermöglicht. Die Speicherbereinigung erfolgt nebenläufig (im System-Thread) nach dem Mark-Compact-Verfahren mit dem Dreifarb-Markierungsalgorithmus. Zwei Implementierungen sind wählbar:

1. Programmspeicherverbrauchsoptimiert, es werden einfache Suchalgorithmen bei der Kompaktierung verwendet.

2. Durchsatzoptimiert, die Speicherblöcke werden in einer nach Adressen sortierten Liste verwaltet, die Blocksuche bei der Kompaktierung entfällt.

Thread-Scheduling

Der Thread-Scheduler, teilt die Rechenzeit den Java-Threads (nebenläufigen Programmfäden) zu. Die Zuteilung erfolgt zeitgesteuert im Raster einer Millisekunde. Jedem Thread ist eine Ausführungswahrscheinlichkeit (eine Java-Priorität von 1 bis 10) und ggf. eine Zeitschranke (Deadline) im Bereich von 0 bis 32767 ms zugeordnet. Die Zuteilung der Rechenzeit erfolgt in drei Schichten:

1. Alle nicht blockierten Threads mit der dringlichsten Zeitschranke sind ausführbar (Earliest Deadline First; EDF).

2. Alle ausführbaren Threads werden entsprechend ihrer Ausführungswahrscheinlichkeit behandelt (zählerbasierte Zuteilung dynamischer Prioritäten; Fair-Share-Scheduling).

3. Alle ausführbaren Threads mit derselben Ausführungswahrscheinlichkeit werden nacheinander aufgerufen (Round-Robin).

Der Scheduler ist unterbrechend und bevorrechtigt (preemptive). Die Threadausführung wechselt nach maximal 64 ms (Zeitscheibe). Externe oder interne Ereignisse verringern diese Zeit. Threads, die ihren Wartezustand verlieren, werden einmalig bevorzugt (Priority-Boost). Die zum Wechsel der Threads benötigte Zeit ist linear abhängig von der Zahl der vorhandenen Threads und liegt typisch zwischen ca. 300 μs (ein Thread) und 800 μs (acht Threads). Sie liegt damit für die meisten Anwendungsfälle unterhalb der Entscheidungsschwelle von 1 ms. Der Scheduler wird nur an Bytecode-Grenzen und außerhalb von Blöcken mit nativen Code aufgerufen, dies vermeidet zeitweise inkonsistente Speicherbereiche. Für die Integration der Zeitschranken existiert eine Spracherweiterung von Java. Zeitvorgaben werden in ein `Deadline`-Objekt gekapselt und dies zeitkritischen Code-Abschnitten mit dem `synchronized`-Schlüsselwort zugeordnet.

Programmspeicher

Java-Anwendungen werden von speziellen Entwicklungswerkzeugen vorverlinkt und zu einem Abbild (Image) zusammengestellt. Hierbei findet auch eine Redundanzoptimierung der Klassendateien statt. Im Fall des internen Programmspeichers wird dabei vorhandener nativer Code in das Abbild integriert (Inlining).

Unterschiede zur Spezifikation

Die Yogi2-VM orientiert sich an der Spezifikation der virtuellen Maschine für die J2SE und die J2ME, nicht an der für kleinere Systeme, wie der Java Card. Einige Komponenten verhalten sich auf der Yogi2-VM jedoch anders, als in der Spezifikation vorgeschrieben:

- Nicht unterstützte Java-Datentypen: `long`, `float` und `double`, damit verbundene Bytecodes sind nicht realisiert.
- Der Datentyp `int` verwendet nur 16 statt 32 Bit.
- Der Konstantenpool in den Klassendateien darf nach der Vorverlinkung 255 Einträge nicht überschreiten.
- Arrays haben nur eingeschränkte Funktionalität: Es ist nicht möglich, Methoden aus `java.lang.Object` darauf anzuwenden. Ein Cast-Check von Arrays primitiver Datentypen wird nicht unterstützt.
- Einige Ausnahmen (Exceptions) werden in der VM vereinfacht bearbeitet und sind nicht behandelbar, stattdessen steht ein Error-Handler zur Verfügung, der nach einem Neustart aufgerufen wird.
- Die Speicherbereinigung unterstützt nicht den Aufruf ggf. vorhandener Finalizer, die mit der Methode `finalize()` deklariert wurden.
- Die Klassen des fundamentalen Pakets `java.lang` sind in ihrer Anzahl reduziert und verfügen über reduzierte Eigenschaften und Funktionalität. Dies ist abhängig von der verwendeten Laufzeitbibliothek.

Elektrische Eigenschaften

Die elektrischen Eigenschaften entsprechen denen des verwendeten ST7-Mikrocontrollers [27]. In Berarbeitungspausen aller Java-Threads wird der Controller in den leichten Ruhezustand (Wait for Interrupt; `wfi`) versetzt, so dass der Energieverbrauch bezogen auf eine Betriebsspannung von 3,3 V von 50 mW auf ca. 15 mW absinkt.

Anhang B

Quelltextauszüge

Den Vollständigen Quelltext der Yogi2-VM und des *JControl*-APIs hier aufzuführen, würde den Rahmen dieser Arbeit sprengen. Er steht daher unter [Böh06c] zusammen mit einer statischen Kopie der auf Seite 139 verzeichneten Internetquellen zur Verfügung. Einige Quelltext-Auszüge mit Kernkomponenten, die speziell in dieser Arbeit genannt werden, sollen an dieser Stelle dennoch gezeigt werden.

B.1 Beispiel einer nativen Methode

Zum Integrieren nativen Codes ist ein Entwurfsmuster einzuhalten, das den Methodenrumpf beschreibt. Dabei kommen speziell formatierte Kommentare (`;; `) und Platzhalter (`<..>`) zum Einsatz. Ferner existieren spezielle Makros für den Datenaustausch mit der JavaVM (`mFr_`) und zum Einfügen von Bytecode (`mBc_`), die an dieser Stelle nicht näher erläutert werden. Eine Anleitung zum Erstellen von nativen Methoden, deren Einbindung in ein Projekt und dessen Übersetzung zu einen VM-Abbild beschreibt [Böh02].

Das folgende Beispiel zeigt den nativen Code der Methode `jcontrol.io.GPIO .setState(..)` und wurde direkt aus dem Quelltext übernommen, einige Kommentare wurden ergänzt:

```
;; native:  public native static void setState(int ch, boolean st);
            dc.w    0, 2                    ; Stack, Locals
            dc.w    0, l<id>end - l<id>start ; Größe
l<id>start:
            mBc_CallNative          ; auf nativen Code schalten
            mFr_GetLcVars           ; Zeiger auf lokale Var. ermitteln
            mFr_rdLcVar.b  ,x,0     ; Lokale Var. ch in Register x laden
            jrmi           c<class>error
            cp             x,#low(_<class>PinCount)
            jruge          c<class>error
            push           x
            mFr_rdLcVar.b  ,a,1     ; Lokale Var. st in Register a laden
            pop            x
```

```
            call        l<class>set
            mBc_LeaveNative         ; auf Bytecode schalten
            mNat_Return             ; Bytecode RETURN
l<id>end:
            dc.w        0           ; keine Exception-Tabelle
```

B.2 Die Interpreter-Hauptschleife

Die Interpreter-Hauptschleife stellt den kritischen Pfad bei der Ausführung von Java-Bytecode dar. Sie wird daher hier zusammen mit der Implementierung des Bytecodes `NOP` aufgelistet. Es wurden lediglich einige hier irrelevante Quelltextzeilen zum Debugging und zur Konfiguration gekürzt. Die benötigten Taktzyklen des typischen Ausführungspfades wurden am linken Rand ergänzt (ermittelt mit [26]). Dabei wurden folgende Annahmen getroffen:

- Externer Festwertspeicher für Java-Anwendungen wird nicht verwendet. Der zur Fallunterscheidung notwendige Code (im Makro `moveX`) ist nicht aktiv.
- Der seltene Fall eines Übertrags bei der 16-Bit-Addition des `JPC`s wird nicht betrachtet.
- Obwohl nicht notwendig, wird im `NOP`-Bytecode eine native `nop`-Instruktion ausgeführt.

aus Bytecode.asm

```
  ;*****************************************************************************
  ;;  public:     cBc_Interpret
  ;
  ;   startet den Bytecode-Interpreter
  ;
  ;   input:      xBc_JPC                 virtueller Programmzähler zeigt auf nächsten Bytecode
  ;               pFr_lcVarsX             lokale Variablen (bei Rahmen-Selektierung gesetzt)
  ;               pFr_opStackX            Operandenstack (bei Rahmen-Selektierung gesetzt)
  ;               y                       Operandenstackpointer
  ;
  ;   changes:    xBc_JPC                 zeigt auf den nächsten auszuführenden Bytecode
  ;
  ;*****************************************************************************
  cBc_Interpret:  mGbl_StoreSP v8Bc_SP    ; Stapelzeiger sichern
                  jra     lBc_InterprLoop
  ;-----------------------------------------------------------------------------
  ;   hier kehren die Bytecodes zurück
5 cBc_InterprOne: inc     pxBc_JPC+1.b    ; Optimierung für 1-Byte Bytecodes
3                 jrne    cBc_InterprThis
                  jra     lBc_InterprCar  ; quasi-Carry verrechnen
  cBc_InterprNext:add     a,pxBc_JPC+1.b  ; Lowbyte des Programmzählers erhöhen
                  ld      pxBc_JPC+1.b,a  ; und zurückschreiben
                  jrnc    cBc_InterprThis ; kein Übertrag
  lBc_InterprCar: inc     pxBc_JPC.b      ; Highbyte Inkrementieren
                  jra     cBc_InterprThis
  ;-----------------------------------------------------------------------------
  ;   hierhin springt das Makro LeaveNative, auf dem Stack steht der neue JPC
```

```
  cBc_InterprNat: move.w  ,pxBc_JPC,,++s
  ;--------------------------------------------------------------------------
  ;   hierhin kehren die Bytecodes zurück, die den xBc_JPC selbst manipulieren,
  ;   soll die Interpretation fortgesetzt werden?
  cBc_InterprThis:
3                 ld      a,bBc_AbortInterPr         ; ld a ist schneller als tnz
3                 jrne    lBc_AbortInterpretation
  ;--------------------------------------------------------------------------
  ;   hier wird zum aktuellen Bytecode verzweigt
  lBc_InterprLoop:
6                 moveX   ,x,0,xBc_JPC               ; aktuellen Bytecode nach x
5                 ld      a,(tBc_BytecodesL,x)       ; Lowbyte des Sprungziels auf den Stack
3                 push    a
5                 ld      a,(tBc_BytecodesH,x)       ; Highbyte des Sprungziels auf den Stack
3                 push    a
6                 ret                                ; ... und anspringen

(...)

  ;****************************************************************************
  ;;  bytecode:   cBc_nop
  ;****************************************************************************
  ;
  ;   nop, tue nichts
  ;
  ;****************************************************************************
2 cBc_nop:        nop
3                 mBc_ReturnFromBC 1      ; normal weiter
```

Anhang C

Datenstrukturen

Die Datenstrukturen der Yogi2-VM finden sich in einer frühen Fassung in [Böh01], sie wurden hier erneut zusammengestellt und aktualisiert. In den Tabellen werden bestimmte Datentypbezeichner werwendet: `p` für Zeiger (Pointer) und `v` für Werte (Values). Wenn nicht extra angegeben, wird die natürliche Wortbreite des Zielsystems (16 Bit) verwendet, andernfalls ist die verwendete Bitzahl angehängt. Indexangaben finden hingegen auf Basis von Bytes statt.

C.1 Der Heap

Der Heap hält alle Java-Laufzeitdatenstrukturen (definiert in `Heap.inc`), er wird von vier Zeigern begrenzt:

Variable	Beschreibung
`_Hp_BOTTOM`	untere Begrenzung des Heaps (`$180`)
`pHp_Pointer`	untere Begrenzung des freien Bereichs des Heaps, wird bei einer Speicheranforderung erhöht)
`pHp_Handles`	zeigt auf den untersten belegten Handle (die Differenz aus `pHp_Handles` und `pHp_Pointer` ergibt den freien Speicher)
`_Hp_TOP`	zeigt auf das erste Byte oberhalb des Heaps (`$c00`, variabel)

C.1.1 Handletabelle

Die Handletabelle wächst von oben (`_Hp_TOP`) nach unten, sie besteht aus 12-Bit-Zeigern auf Blöcke im Heap; freie Handles werden durch `null`-Zeiger angegeben. Die übrigen 4 Bit werden ausmaskiert und stehen der Speicherbereinigung zur Verfügung:

Bitmaske	Name	Beschreibung
`%gg****** ********`	`kHp_Garbage`	Maske, um Flags des Garbage-Collectors zu maskieren
`%****pppp pppppppp`	`kHp_NGarbage`	Maske, um Blockzeiger zu maskieren

Bitmaske	Name	Beschreibung
%gggg**** ********	kHp_Garbage	Maske, um alle GC-Flags zu maskieren
%00****** ********	kHp_GarbWhite	markiert Block weiß
%01****** ********	kHp_GarbGray	markiert Block grau
%10****** ********	kHp_GarbBlack	markiert Block schwarz (belegt)
%11****** ********	kHp_GarbComp	markiert Block als kompaktiert
%**f***** ********	kHp_GarbFinal	markiert Block als noch zu finalisieren (z. Zt. ungenutzt)
%***i**** ********	kHp_GarbInvalid	markiert Block als ungültig (geteiltes Flag)
%***l**** ********	kHp_GarbLocked	markiert Block als locked (unbeweglich, geteiltes Flag)

Geteilte Flags treten nur in unterschiedlichen Zuständen der Speicherbereinigung auf.

C.1.2 Heap-Blöcke

Alle Heap-Entries (Blöcke, die über die Handletabelle adressiert werden) haben beim negativen Offset einen Header:

Offset	Name	Typ	Beschreibung
-3	iHp_LengthHigh	v	nur bei Blöcken >255 Bytes
-2	iHp_Length	v8	nur bei Blöcken >14 Bytes
-1	iHp_BlockFlags	v8	%otttllll
0	iHp_BlockData[Hp_Length]	–	die eigentlichen Nutzdaten

Die beiden `Length`-Bytes sind dabei optional (je nach Größe des Blocks), die `BlockFlags`, die u. a. den Block-Typ bestimmen, sind hingegen obligatorisch:

Flag-Maske	Name	Beschreibung
%ottt****	kHp_Type	Typ-/Objektbezeichner
%0000****	kHp_Free	freier/freigegebener Block
%0001****	kHp_Bools	Boolean-Array
%0010****	kHp_Bytes	Byte-Array
%0011****	kHp_Words	Word-Array
%0100****	kHp_References	Objekt-Array
%0111****	kHp_Init	Referenz auf den Namen einer zu initialisierenden Klasse
%1000****	kHp_Object	Objekt-Block
%1001****	kHp_Class	Klassen-Objekt
%1010****	kHp_Thread	Thread-Objekt
%1011****	kHp_String	String-Objekt
%1100****	kHp_Deadline	Deadline-Objekt
%1101****	kHp_Atomic	Atomic-Object (reserviert)
%1110****	kHp_Memory	Memory-Object (reserviert)
%1111****	kHp_Unspec	unspezifizierter Block

Die Längenangabe wird dabei wie folgt kodiert:

Flag-Maske	Beschreibung
`%****llll`	direkte Längenangabe (kein `Hp_Length`):
`%****0001`	Blocklänge: 1
⋮	⋮
`%****1110`	Blocklänge: 14
`%****0000`	8-Bit-Längenangabe in `Hp_Length`
`%****1111`	16-Bit-Längenangabe in `Hp_LengthHigh|Hp_Length`

C.1.2.1 Reguläre Heap-Blöcke

Zur Geschwindigkeitsoptimierung der Kompaktierung des Heaps wurde in Abschnitt 5.3.1 eine alternative Speicherverwaltung mit einem modifiziertem Header eingeführt (die Flags entsprechen den oben gezeigten, die Blocklänge wird aus der Adressdifferenz zum nächsten Block berechnet):

Offset	Name	Typ	Beschreibung
-2	`iHp_NextLow`	$p8_l$	Lowbyte des Zeigers auf den Handle des nächsten Blocks
-1	`iHp_BlockFlags`	$v4/p4_h$	`%otttpppp` Flags und obere 4 Bit des Zeigers
0	`iHp_BlockData[Hp_Length]`	–	die eigentlichen Nutzdaten

C.2 Block-Typen

Im Folgenden werden die Strukturen beschrieben, die hinter den Block-Headern liegen (`Hp_BlockData[Hp_Length]`), sie werden direkt durch einen Handle adressiert.

C.2.1 Der `Object`-Block

Alle Blöcke vom Typ Objekt sind mit einem Typbezeichner markiert, dessen oberstes Bit (`%1***`) gesetzt ist, und verfügen über einen speziellen Objekt-Header (definiert in `Objects.inc`):

Offset	Name	Typ	Beschreibung
0	`iObj_Class`	`p`	Handle der Klasse des Objekts
2	`iObj_Monitor`	`p`	Handle des Threads, der den Monitor erworben hat
4	`iObj_MonCount`	`v8`	Zähler für der Anzahl der Eintritte in den Monitor

Offset	Name	Typ	Beschreibung
5	`iObj_Extension`	`v8`	Index innerhalb dieser Struktur auf den Erweiterungsblock (zeigt hinter die Variablen – `null`, wenn kein Erweiterungsblock)
6	`iObj_IFields`	`[]`	Instanzvariablen (`Cls_IFieCnt`)
`Obj_Extension`		`[]`	optionaler Erweiterungsblock

Die Anzahl der Einträge in der Variablentabelle wird dabei aus der referenzierten Klassenstruktur ermittelt.

C.2.1.1 Der `Class`-Block

Eine Klasse ist eine Unterstruktur eines Objekts, nur dass dort, wo sich im Objekt (also einer Klassen-Instanz) die Instanzvariablen (`ifields`) befinden, die (statischen) Klassenvariablen zu finden sind (`sfields`); der Monitor wird ebenfalls als Klassenmonitor verwendet, die Klassenreferenz im Klassen-Objekt zeigt auf sich selbst zurück. Die Tabelle zeigt nur den Erweiterungsblock (definiert in `Class.inc`):

Offset	Name	Typ	Beschreibung
-2i	–	`p[]`	Laufzeitreferenztabelle (negativer Offset)
0	`iCls_This`	`v8`	diese Klasse (Index im Konstantenpool, direkt auf Utf8-Zeichenkette)
1	`iCls_Super`	`v8`	Oberklasse (Index im Konstantenpool, auf die Klassen-Referenz)
2	`iCls_Flags`	`v8`	Flags der Klasse (Memtype ...)
3	`iCls_PoolCnt`	`v8`	Anzahl der Einträge im Konstantenpool
4	`iCls_DFieCnt`	`v8`	Anzahl der deklarierten Datenfelder
5	`iCls_SFieCnt`	`v8`	Anzahl der Klassenvariablen insgesamt
6	`iCls_IFieCnt`	`v8`	Anzahl der Instanzvariablen insgesamt
7	`iCls_DMethCnt`	`v8`	Anzahl der deklarierten Methoden
8	`iCls_MethCnt`	`v8`	Anzahl der Methoden insgesamt
9	`iCls_InterfCnt`	`v8`	Anzahl der implementierten Interfaces
10	`iCls_Constants`	`p`	Zeiger auf externe CPR
12	`iCls_DFields`	`p`	Zeiger auf externe Tabelle mit deklarierten Variablen
14	`iCls_DMethods`	`p`	Zeiger auf externe Tabelle mit deklarierten Methoden
16	`iCls_Methods`	`p`	Zeiger auf externe Methoden-Code Referenztabelle
18	`iCls_Interfaces`	`p`	Zeiger auf externe Interfaces Referenztabelle

Wichtige Informationen über den Aufbau der Klassen-Struktur ist dem `Cls_Flags`-Eintrag zu entnehmen:

Bitmaske	Name	Beschreibung
`%m*******`	`kCls_MemType`	Maske, um Speichertyp der Klasse zu maskieren
`%0*******`	`kCls_MemType`	Klasse liegt im internen Speicher (ROM)
`%1*******`	`kCls_MemType`	Klasse liegt im externen Speicher (Flash)

Bitmaske	Name	Beschreibung
%*1******	kCls_DeclaredRefs	Datenfeld- und Methoden-Referenztabellen nicht aufgelöst, pxCls_DFields und pxCls_SMethods sind dann nur einen Eintrag groß und weisen auf die erste Variblen- bzw. Methodendeklaration in der Klassendatei (bei pxCls_DMethods entfällt auch der Eintrag am negativen Offset), kommt in der aktuellen Realisierung nicht mehr vor
%**1*****	kCls_ExtStructs	die Klasse ist vorverlinkt, dann befinden sich bei iCls_Structures nicht die Strukturen selbst, sondern lediglich Zeiger auf diese Strukturen im Festwertspeicher (dann ist kCls_DeclaredRefs irrelevant), kommt in der aktuellen Realisierung nicht mehr vor
%****1***	kCls_KeepInMem	die Klasse wird von der Speicherbereinigung ausgenommen (z. B. nach der Initialisierung)
%*******X	kCls_Public	Klassenmodifikator: ACC_PUBLIC
%***X****	kCls_Final	Klassenmodifikator: ACC_FINAL
%******X*	kCls_Interface	Klassenmodifikator: ACC_INTERFACE
%*****X**	kCls_Abstract	Klassenmodifikator: ACC_ABSTRACT

Die letzten vier Flags werden aus den Flags der Klassendatei übertragen, indem der 16-Bit-Wert access_flags zunächst mit (ACC_PUBLIC+ACC_FINAL+ACC_INTERFACE+ACC_ABSTRACT) und-verknüpft wird und dann die beiden Bytes überlagert werden (oder-verknüpft), somit können die Klassen-Flags später schneller abgefragt werden.

Die Tabelle mit den Laufzeitrefenzen der Klasse befindet sich *vor* der eigentlichen Klassenstruktur und hinter den statischen Variablen der Klasse, die Referenzen können daher von der Speicherbereinigung wie statische Variablen behandelt werden. Indiziert werden die Laufzeitreferenzen mit einem negativen Offset (so wie die Flag-Felder einiger Unterstrukturen auch). Der Eintrag ist auf zweierlei Arten kodiert:

Offset	Typ	Beschreibung
falls Klassenobjekt initialisiert:		
0	v4/p_h	4-Bit-Zugriffszähler, wird beim Zugriff auf 15 gesetzt und vom GC heruntergezählt, bei 0 wird der Eintrag in die andere Form gewandelt; ermöglicht das verzögerte Entladen von Klassen; obere 4 Bit des Handles der Klasse
1	p_l	untere 8 Bit des Handles der Klasse
falls Referenz in auf CPR:		
0	–	0, Erkennungsmerkmal
1	v8	Index in der CPR mit Referenz auf CONSTANT_CLASS-Eintrag, wird zum Initialisieren der Klasse verwendet

Klassenstrukturen im Festwertspeicher

Das Klassenobjekt auf dem Heap verweist auf Strukturen im Festwertspeicher (abhängig von `Cls_MemType` im internen ROM oder externen Flash-Speicher). Sie werden in Anhang C.3.2.1 aufgelistet.

C.2.1.2 Der `Thread`-Block

Bei Threads handelt es sich im Gegensatz zu den Klassen um echte Objekte mit Erweiterungsblock. Der System-Thread nutzt den Objekt-Teil nicht (für den System-Thread gibt es kein Objekt, die Klasse `java.lang.Thread` muss nicht geladen werden, ja noch nicht einmal vorhanden, sein). Die Tabelle zeigt nur den Erweiterungsblock (definiert in `Treads.inc`):

Offset	Name	Typ	Beschreibung
0	`iThr_Flags`	v8	Flags des Threads
1	`iThr_Flags2`	v8	weitere Flags
2	`iThr_Pri`	v8	Priorität des Threads
3	`iThr_PriCnt`	v8	laufender Prioritätszähler
4	`iThr_Wait`	v	Zeitpunkt, bis zu dem der Thread wartet, oder Handle des blockierenden Threads
6	`iThr_DLine`	v	Zeitschranke des Threads
8	`iThr_StackLen`	v8	die Länge einer Rahmenstapel-Hälfte (max. 255)
9	`iThr_StackPtr`	v8	aktueller Operandenstapelzeiger
10	`iThr_AktFrame`	v8	Index des aktuellen Rahmens im Stapel
11	`iThr_Next`	p	Handle des nächsten Threads in der zirkularen Liste
13	`iThr_Stack`	`[Thr_StackLen]`	der Rahmenstapel (highbyte)
–	–	`[Thr_StackLen]`	der Rahmenstapel (lowbyte)

Die Größe des Rahmenstapels ergibt sich aus 2·`Thr_StackLen`, kann aber auch implizit durch die Größe des Speicherblocks ermittelt werden (bei der Erzeugung des Threads wird diese festgelegt); `Thr_StackLen` hilft nur, die Zeiger auf den aktuellen Rahmen schneller berechnen zu können. Die Flags eines Threads sind folgende:

Flags	Name	Beschreibung
`%00011111`	`kThr_RunMask`	And-Maske für Run (bei `0` wird der Thread ausgeführt)
`%***00001`	`kThr_Dead`	Thread läuft nicht (wurde beendet)
`%******X*`	`kThr_Wait`	Thread wartet auf externes Ereignis (notify)
`%*****X**`	`kThr_WaitST`	Thread wartet auf den System-Thread
`%****X***`	`kThr_WaitMoni`	Thread wartet auf freiwerdenden Monitor
`%***X****`	`kThr_WaitUpTo`	Thread wartet auf Zeitpunkt
`%**X*****`	`kThr_System`	markiert den System-Thread

Flags	Name	Beschreibung
%*X******	kThr_EDF	verwende EDF-Scheduling
%X*******	kThr_Daemon	Hintergrundthread (kann automatisch beendet werden)
%11100000	kThr_TypeMask	And-Maske für Thread-Typ
%10000000	kThr_NewMask	And-Maske für Übernahme von Flags in neuen Thread
Flags2	**Name**	**Beschreibung**
%X*******	kThr_Interrupt	Interrupted-Flag
%*X******	kThr_SingleStep	Einzelschritt-Modus (Debugger)
%**X*****	kThr_Trace	Trace-Modus (Debugger, Breakpoint)

Rahmen

Die Rahmenstruktur ist eine Unterstruktur eines Threads (im Rahmenstapel), sie ist zweigeteilt in eine High- und eine Low-Struktur (definiert in `Frame.inc`):

Offset	Name	Typ	Beschreibung
FRAME_HIGH:			
-Fr_CntVars	pFr_lcvars	v_h[Fr_CntVars]	lokale Variablen (highbytes)
0	iFr_Class	p_h	Handle der Klasse des Rahmens (highbyte)
1	iFr_ByteCode	p_h	Zeiger auf Bytecodeanfang (highbyte)
2	iFr_PC	p_h	Zeiger auf aktuellen Bytecode (highbyte)
3	iFr_Flags	v8	Flags des Rahmens
4	iFr_opStack	v_h[...]	Operandenstack (highbytes)
FRAME_LOW:			
-Fr_CntVars	pFr_lcvars	v_l[Fr_CntVars]	lokale Variablen (lowbytes)
0	iFr_Class	p_l	Handle der Klasse des Rahmens (lowbyte)
1	iFr_ByteCode	p_l	Zeiger auf Bytecodeanfang (lowbyte)
2	iFr_PC	p_l	Zeiger auf aktuellen Bytecode (lowbyte)
3	iFr_Prev	v8	Index in Thread-Struktur des vorherigen Rahmens
4	pFr_opStack	v_l[...]	Operandenstack (lowbytes)

Die Bitmaske für `Fr_Flags` ist folgende:

Flags	Name	Beschreibung
%00111111	kFr_CntVars	maskiert die Anzahl der lokalen Variablen (um `pxFr_lcVars` ermitteln zu können)
%01000000	kFr_Monitor	maskiert einen freizugebenden Monitor (Handle ist auf dem Stack)
%10000000	kFr_External	maskiert Speichertyp-Bit (erspart nachschauen in der Klasse des Rahmens)

C.2.1.3 Der `String`-Block

Der Erweiterungsblock des String-Objekts enthält als Option (falls keine Referenz auf einen Speicherbereich vorliegt) die Utf8-kodierte Zeichenkette (definiert in `Strings.inc`):

Offset	Name	Typ	Beschreibung
0	`iStr_Flags`	`v8`	Flags
falls intern:			
1	`iStr_Length`	`v16`	Länge der Zeichenkette in Bytes
3	`iStr_Utf8`	`v8[Str_Length]`	Zeichenkette
falls extern:			
1	`iStr_Data`	`p`	Zeiger auf Zeichenkette im Festwertspeicher

Die Flags spezifizieren die Variante des String-Objekts:

Flags	Name	Beschreibung
`%1******`	`kStr_Extern`	String ist extern, d. h. nicht im String-Objekt
`%*1*****`	`kStr_MemType`	Speichertyp des externen Strings (in `iStr_Data`)
`%**1*****`	`kStr_Buffer`	String-Buffer (intern, Größe veränderbar, nicht unterstützt)

C.2.1.4 Der `Deadline`-Block

Objekte vom Typ `Deadline` definieren für die relevanten Informationen entsprechende Java-Datenfelder (Instanzvariablen): `timeconstraint`, `flags` und `oldDeadline`, die Flags sind nur auf der Ebene der virtuellen Maschine definiert und werden ausschließlich von nativem Code benutzt (definiert in `Deadline.asm`):

Flags	Name	Beschreibung
`%*******1`	`k<class>hasOldDeadline`	alte Deadline vorhanden?
`%******1*`	`k<class>DeadlineMiss`	Zeitschrankenüberschreitung aufgetreten?

C.2.2 Der `Init`-Block

Befindet sich ein `Init`-Block auf dem Heap, so wird er vom System-Thread heran gezogen, eine Klasse zu lokalisieren und initialisieren. Da dies nebenläufig geschieht, wird der aktuelle Zustand der Initialisierung im `Init`-Block gesichert.

Offset	Name	Typ	Beschreibung
0	`iCls_LoadFlags`	`v8`	Flags für die Klasseninitialisierung
1	`iCls_LoadState`	`v8`	merkt sich den Fortschritt der Klasseninitalisierung (Zähler)

Offset	Name	Typ	Beschreibung
2	`iCls_LoadHandle`	`p`	Handle der teilweise gefüllten Klassenstruktur (oder der Superklasse, abhängig vom Zustand)
4	`iCls_LoadMethCnt`	`v8`	Anzahl der Einträge in der Methodenreferenztabelle
5	`iCls_LoadName`	`p`	Zeiger auf Utf8-Zeichenkette mit Klassennamen
optional, abhängig von den Flags:			
7	`iCls_LoadRunName`	`p`	Zeiger auf Utf8-Zeichenkette mit Namen einer zu startenden Methode
9	`iCls_LoadRunDesc`	`p`	Zeiger auf Utf8-Zeichenkette mit dem Deskriptor der Methode
11	`iCls_LoadParams`	`v[]`	Liste mit den Parametern für den Methodenaufruf (16-Bit-Einträge)
optional, abhängig von den Flags:			
7	`iCls_LoadThread`	`p`	Klasse mit dem mit diesem Handle angegebenen Thread initialisieren

Folgende Flags sind definiert:

Flags	Name	Beschreibung
`%*******1`	`kCls_LoadNameExt`	Speichertyp-Bit für `LoadName`
`%******1*`	`kCls_LoadRun`	eine Methode ist zu starten, wenn die Klasse geladen wurde (dann hat die `Init`-Struktur die `LoadRun`-Zusatzeinträge)
`%*****1**`	`kCls_LoadRunThread`	die Methode wird in einem neu eingerichteten Thread gestartet (mit Standard-Thread-Parametern), sonst läuft sie im System-Thread
`%****1***`	`kCls_LoadRunStatic`	zu startende Methode ist statisch (`ACC_STATIC`)
`%***1****`	`kCls_LoadRunNameExt`	Speichertyp-Bit für `LoadRunName`
`%**1*****`	`kCls_LoadRunDescExt`	Speichertyp-Bit für `LoadRunDesc`
`%*1******`	`kCls_LoadNameRelative`	`LoadName` ist eine blockrelative Adresse (Zeichenkette ist an die `Init`-Struktur angehängt)
`%1*******`	`kCls_LoadInitThread`	die Klasse wird in einem Thread bei `LoadThread` initialisiert (dann kein `LoadRun` möglich)

Die Zustände der Klasseninitialisierung:

Zustand	Name	Beschreibung
0	`_Cls_LoadStateFirst`	Neu angelegter `Init`-Block
1	`_Cls_LoadStateMem`	bereit zum Anfordern des Heap-Speichers (nicht bei vorverlinkten Klassen)
2	`_Cls_LoadStateFill`	bereit zum Füllen der Laufzeitstruktur (nicht bei vorverlinkten Klassen)
2	`_Cls_LoadStateNoFile`	Zeiger auf Klassen-Abbild muss nicht mehr ermittelt werden (Schwellwert)
3	`_Cls_LoadStateFinished`	Klassenstruktur vollständig (impliziter Zustand, `Init`-Struktur entfernt)

Da in der aktuellen Realisierung nur noch vorverlinkte Klassen unterstützt werden, kommen die Zustände `LoadStateMem` und `LoadStateFill` nicht mehr vor.

C.3 Strukturen im Festwertspeicher

Folgende Strukturen werden vom Vorverlinker erzeugt und im Festwertspeicher des Zielsystems abgelegt. Die JavaVM greift nur lesend darauf zu.

C.3.1 Segmentliste

Die Segmentliste befindet sich am Anfang einer Flash-Speicher-Bank und legt durch die Reihenfolge ihrer Einträge die Suchreihenfolge der Archive für die VM fest (definiert in `ROMfile.inc`):

Offset	Name	Typ	Beschreibung
0	`iJCA_SegmentMagic`	`v32`	Magic-Number: „`JCSL`“
4	`iJCA_SegmentVersion`	`v`	Versionsnummer (nur vom Vorverlinker verwendet)
6	`iJCA_SegmentCheck`	`v`	Prüfsumme der Bank (nur vom Vorverlinker verwendet)
8	`iJCA_SegmentMainName`	`p`	Zeiger auf Utf8-Zeichenkette mit Namen der Bank (optional)
10	`iJCA_SegmentMainFlags`	`v`	Zugriffsflags der Bank
12	`iJCA_SegmentFree`	`v`	Anzahl der freien User-Sektoren

Die einzelnen archivbeschreibenden Segmente schließen sich direkt an diese Struktur an:

Offset	Name	Typ	Beschreibung
0	`iJCA_Segment`	`p`	Zeiger auf Segment mit Archiv (-1: Ende der Liste; 0: Verweis auf ROM-Archiv)
2	`iJCA_SegmentLen`	`v`	Länge des Archivs (in Bytes; beim ROM-Verweis irrelevant)
4	`iJCA_SegmentName`	`p`	Zeiger auf Utf8-Zeichenkette mit Namen des Archivs (optional, sonst `null`)
6	`iJCA_SegmentFlags`	`v`	Zugriffsflags für einzelnes Archiv

Die Zeiger auf die Banknamen können dabei in den freien Bereich zwischen der Segmentliste und dem ersten Archiv weisen oder vorhandene Zeichenketten in den Archiven nutzen. Die Zugriffsflags können vom Vorverlinker gesetzt werden, um versehentliches Löschen von Bänken oder Archiven zu verhindern. Folgende Flags sind definiert:

Flags	Name	Beschreibung
%*******1	kJCA_SegmentFlagNotWriteable	/w, Schreibschutz
%******1*	kJCA_SegmentFlagNotReadable	/r, Ausleseschutz (betrifft den Vorverlinker, auf dem Zielsystem wird nur die Erstellung eines Inhaltsverzeichnisses verhindert)
%*****1**	kJCA_SegmentFlagSystem	s, markiert Systemarchiv/-bank (z. B. SystemSetup)

C.3.2 Archive

Jedes Archiv besteht aus einer verketteten Liste, deren Einträge auf die einzelnen Ressourcen oder Klassen-Abbilder verweisen. Die Position der Nutzdaten ist dabei nicht vorgeschrieben, üblicherweise folgen sie direkt auf die entsprechenden Archiv-Einträge. Um die Verwaltung durch den Vorverlinker zu erleichtern, sollten alle Elemente eines Archivs in einem zusammenhängenden Speicherbereich geschrieben werden, dessen Größe in der Segmentliste unter `JCA_SegmentLen` eingetragen wird:

Offset	Name	Typ	Beschreibung
0	iJCA_NextEntry	p	Zeigt auf den nächsten Eintrag (verkettete Liste), `null` für Listenende
2	iJCA_EntryName	p	Zeigt auf die Utf8-Zeichenkette mit Dateinamen (inkl. Längenangabe), kann auch innerhalb der Nutzdaten stehen
4	iJCA_Entry	p	Zeigt auf die Nutzdaten (Ressource oder Klassenabbild)
6	iJCA_Comment	p	Zeigt auf Zusatzdaten (z. B. die Tabellen der vorverlinkten Klassen, nicht bei Ressourcen)
8	iJCA_EntryLen	v	Länge der Nutzdaten (negativ bei Klassenabbildern, unterscheidet sie von Ressourcen)
10	iJCA_CommentLen	v	Länge der Zusatzdaten (nur informativ)

C.3.2.1 Klassenabbilder

Klassenabbilder teilen sich in den Kommentarbereich (Zusatzdaten) und den eigentlichen Klassenbereich (Nutzdaten). Der Kommentarbereich enthält alle vom Vorverlinker neu erzeugten Strukturen (definiert in `Class.inc`):

Offset	Name	Typ	Beschreibung
0	_CC_Magic	v32	Magic-Number: `$01cecafe`
4	_CC_Length	v	Länge der Mirror-Struktur
6	_CC_Mirror	v8[_CC_Length]	Abbild des `Class`-Blocks, der bei der Initialisierung auf den Heap kopiert wird

Der `Class`-Block wurde bereits in Anhang C.2.1.1 beschrieben, die darin enthaltenen Zeiger weisen in die Datenstrukturen dieses Kommentarbereichs.

`Cls_Constants` verweist auf die CPR, sie enthält eine Zusammenfassung des Konstantenpools der Klassendatei und wurde bereits in Abschnitt 4.2.3.1 beschrieben. Die von der CPR referenzierten Elemente des gekürzten Konstantenpools befinden sich in den Nutzdaten eines beliebigen Klassenabbilds innerhalb dieses Archivs. Abgesehen von der Entfernung von Einträgen, die nicht benötigt werden oder mehrfach (in allen Klassen) vorkommen, entspricht das Format der Konstantenpool-Einträge dem in [LY00] definierten:

Offset	Typ	Beschreibung
-i	`v8`	Tag, muss negativ sein, indiziert bei Klassen-, Methoden- oder Datenfeldreferenzen die Laufzeitreferenzentabelle
falls numerische Konstante:		
2i	`v`	konstanter Wert
falls Zeichenkette, Klasse oder String:		
2i	`p`	Zeiger auf Konstantenpool-Eintrag mit Utf8-Zeichenkette
falls Methoden- oder Datenfeldreferenz:		
2i	`v8`	Index in der entprechenden Tabelle (MRT oder Datenfeld)
2i+1	`v8`	Index in dieser CPR mit Name-and-Type-Eintrag
falls Name-and-Type-Eintrag:		
2i	`v8`	Index in dieser CPR mit Utf8-Zeichenkette des Namens
2i+1	`v8`	Index in dieser CPR mit Utf8-Zeichenkette des Deskriptors

`Cls_DFields` verweist auf die Liste `field_info fields[Cls_DFieCnt]`, welche in [LY00] definiert ist und die auf die entsprechenden `field_info`-Strukturen im Nutzdatenbereich dieses Klassenabbilds verweist. Da ein symbolischer Zugriff auf Datenfelder nicht unterstützt wird, wurde sie vom Vorverlinker gekürzt:

Offset	Typ	Beschreibung
2i	`p`	Zeiger auf `field_info`

`Cls_DMethods` verweist analog auf die Liste `method_info methods[Cls_DMethCnt]`. Auch diese Liste wurde gekürzt und enthält nur noch Einträge, die zum Aufruf von Methoden nötig sind, von denen nur symbolische Informationen vorliegen (z. B. `main(..)`). Am negativen Index enthält sie Verweise auf die entsprechenden Einträge der MRT:

Offset	Typ	Beschreibung
-i	`v8`	Index der entsprechenden Methode in der MRT
2i	`p`	Zeiger auf `method_info`

`Cls_Methods` verweist auf die Methodenreferenztabelle (siehe Abschnitt 4.2.3.2):

Offset	Typ	Beschreibung
-i	`v8`	Flag
falls in dieser Klasse definiert:		
$2i$	`p`	Zeiger auf `Code`-Attribut der Methode
falls in anderer Klasse definiert:		
$2i$	`v8`	Index in anderer MRT
$2i+1$	`v8`	Index in dieser CPR mit `CONSTANT_CLASS`-Eintrag (dessen MRT wird indiziert)

Folgende Flags sind definiert (aus `Class.inc`):

Flags	Name	Beschreibung
`%****1111`	`kCls_MethParams`	maskiert die Anzahl der Parameter (→ max 15)
`%***1****`	`kCls_MethInh`	Methode ist vererbt
`%**1*****`	`kCls_MethSync`	Methode ist `synchronized`
`%*1******`	`kCls_MethCode`	`Code` schon gefunden und eingetragen
`%1*******`	`kCls_MethMemType`	Speichertyp der Methode

`Cls_Interfaces` verweist auf die aufgelöste Liste aller von dieser Klasse implementierten Interfaces:

Offset	Typ	Beschreibung
-i	`v8`	Startindex in der MRT der implementierenden Methoden
i	`v8`	Index in dieser CPR mit `CONSTANT_CLASS`-Eintrag der Interfaceklasse

Im Nutzdatenbereich befinden sich nur noch die Strukturen der Klassendatei, die von den Strukturen im Kommentarbereich direkt oder indirekt (z. B. über einen Konstantenpool-Eintrag) referenziert werden. Alle übrigen Elemente sind redundant oder irrelevant und werden auf dem Zielsystem nicht mehr benötigt. Die in der Spezifikation [LY00] vorgegebene Struktur ist nicht mehr vorhanden.

C.4 Fehlercodes

Die VM-internen Fehlercodes sind implementierungsabhängig. Um dennoch eine Konstanz zu erreichen, wurden im Entwicklungsfluss neue Fehlercodes immer am Ende der Liste angefügt. Ggf. nicht mehr benutzte Fehlertypen blieben in der Tabelle stehen. Das trifft beispielsweise auf alle Fehlercodes zu, die durch die Verwendung eines Vorverlinkers mit einer Konsistenzprüfung des Projekts nicht mehr auftreten können.

ID	Name	Beschreibung
1	`HandleError`	fehlerhafte Referenz in der Handletabelle
2	`NullPointerException`	`null` wurde benutzt, aber eine Objektreferenz benötigt
3	`OutOfMemoryError`	Speicherüberlauf im Heap

ID	Name	Beschreibung
4	`BytecodeNotAvailableError`	nicht implementierter Bytecode (in der Entwicklungsphase)
5	`BytecodeNotSupportedError`	nicht unterstützter Bytecode (z. B. Fließkommaberechnungen)
6	`BytecodeNotDefinedError`	nicht definierter Bytecode (in der Spezifikation)
7	`ArithmeticException`	Fehler bei arithmetischen Berechnungen (z. B. Teilung durch Null)
8	`NegativeArraySizeException`	Versuch ein Array mit negativer Größe zu erzeugen
9	`UnsupportedArrayTypeError`	nicht unterstützter Array-Typ (z. B. `long`)
10	`ArrayIndexOutOfBoundsException`	Zugriff über die Arraygrenze hinaus
11	`ClassCastException`	wird vom `checkcast`-Bytecode ausgelöst
12	`NoCodeError`	leerer Methodenrumpf (`abstract` oder nicht implementierter nativer Code)
13	`WaitForMonitorSignal`	wird intern von der VM verwendet (kein Fehler)
14	`ExternalNativeError`	native Methode außerhalb des Controller-ROMs
15	`FatalStackFrameOverflowError`	Versuch, ein `Thread`-Objekt auf mehr als 256 Stapeleinträge zu erweitern
16	`InstantiationException`	Versuch, eine Instanz einer abstrakten Klasse oder eines Interface zu erzeugen
17	`IllegalMonitorStateException`	Aufruf eines `wait()` ohne im Kontext eines passenden Monitors zu stehen
18	`UnsatisfiedPrelinkError`	fehlerhafte Vorverlinkung festgestellt
19	`ClassFormatError`	Fehler beim lokalen Verlinken einer Klasse (Struktur)
20	`ClassTooBigError`	Fehler beim lokalen Verlinken einer Klasse (Größe)
21	`PreLinkError`	fehlerhafte Vorverlinkung festgestellt
22	`PreLinkedUnresolvedError`	fehlerhafte Vorverlinkung festgestellt (Konstantenpooleintrag nicht aufgelöst)
23	`UnsupportedConstantTypeError`	nicht unterstützter Konstantenpooleintrag (`long`, `float`, `double`)
24	`MalformatedDescriptorError`	Fehlerhafter Klassen- oder Methodendescriptor
25	`RuntimeRefTableOverrunError`	Überlauf der Laufzeitreferenzen-Tabelle einer Klasse
26	`NoSuchFieldError`	Referenziertes Datenfeld nicht vorhanden
27	`IllegalAccessError`	Zugriffsbeschränktes Element (z. B. `private`) im falschen Kontext abgefragt
28	`NoSuchMethodError`	Referenzierte Methode nicht vorhanden
29	`TooMuchParametersError`	Eine Methode benutzt zuviele Parameter (max. 16 unterstützt)
30	`ThrowFinalError`	eine anwendungsdefinierte Ausnahme wurde nicht bearbeitet. Der Name der Ausnahme wird dem `ErrorHandler` übergeben
31	`NoClassDefFoundError`	namentlich bekannte Klasse nicht vorhanden
32	`IndexOutOfBoundsException`	Zugriff über Array- oder Stringgrenzen hinaus

ID	Name	Beschreibung
33	`ArrayDimensionError`	Versuch, ein Array mit mehr als 2 Dimensionen anzulegen
34	`DeadlockError`	Der Thread-Scheduler hat einen Zyklus abhängiger Threads festgestellt
35	`IncompatibleClassChangeError`	Versuch, eine Interfacemethode in einer nicht implementierenden Klasse aufzurufen
36	`NotImplementedError`	Nicht unterstützte Java-Eigenschaft wurde verwendet
37	`WatchdogError`	Durch eine Verzögerung in der VM wurde der Watchdog des Mikrocontrollers ausgelöst
38	`DoubleDIDError`	Innerhalb eines CAN-Netzwerks wurde ein anderes Gerät mit derselben Geräteadresse gefunden

Anhang D

Erweiterungen und Verfeinerungen

Für die hier vorgestellte Yogi2-VM wurden noch Erweiterungen und Verbesserungen vorgesehen, die aber aus implementierungsabhängigen Gründen nur mit großem Aufwand hätten umgesetzt werden können. Das in Kapitel 10 vorgestellte neue Konzept kann durch den objektorientierten Ansatz von Java diese Änderungen leicht integrieren und bietet mehr Raum für Varianten. Diese Erweiterungen sollen hier nur kurz vorgestellt werden (Änderungen und Ergänzungen der Anwendungsprogrammierschnittstelle werden dabei ausgespart):

Bytecode Interpreter

- Abbildung nativer (zielsystemabhängiger) Interrupts auf entsprechenden Java-Code (dafür Unterstützung nicht unterbrechbarer Code-Sequenzen).
- Reduktion der Abfragen der Unterbrechungsflags (evtl. nur bei Sprüngen und Methodenaufrufen) zur Steigerung der Ausführungsgeschwindigkeit; das wird konfigurierbar ausgelegt. Dabei ist allerdings die Reaktivität des Systems bzgl. externer Ereignisse (Interrupts), die nun größtenteils mit Java behandelt werden sollen, zu beobachten.

Speicherverwaltung

- Unterstützung von reservierbaren Speicherbereichen je Thread oder Code-Block, die im Heap exklusiv vorgehalten werden. Das ist eine einfache Form der in [24] spezifizierten abgegrenzten Speicherbereiche (*Scoped Memory*) und nur bei der in Kapitel 5 verwendeten Speicherarchitektur sinnvoll, die eine schnelle Allokation von Blöcken erlaubt. Eine Unterstützung durch Entwicklungswerkzeuge ist erforderlich, um ein Überlaufen dieser reservierten Speicherbereiche zur Laufzeit zu verhindern (bzw. um ihre erforderliche Größe zu bestimmen). Der Prozess kann auch ohne notwendigen Entwicklereingriff für Code-Blöcke mit einer zugeordneten Zeitschranke (Deadline) ablaufen.

- Optimierung der Speicherbereinigung. In der neuen Architektur aus Kapitel 10 können anwendungsspezifisch geeignete Bereinigungsalgorithmen ausgewählt werden. Aber auch die bereits in der Yogi2-VM realisierte Speicherbereinigung kann mit einer adaptiven Rechenzeitsteuerung versehen werden, die sich an der aktuellen Allokationsrate orientiert (siehe Abschnitt 5.3).

Thread-Scheduling

- Die globale Zeitsteuerung mittels eines Millisekunden-Timers wird durch einen frei programmierbaren Timer ersetzt, das spart etwas Overhead ein und ermöglicht prinzipiell auch feinere Abstufungen auf leistungsfähigeren Systemen.

Datenstrukturen

- Ressourcen werden nicht mehr in einer simplen verketteten Liste gehalten, sondern in einer höher geordneten Struktur, um die Suchzeit zu verringern.

- Klassen werden nicht mehr mit ihrem Namen in den Ressourcen gesucht, sondern mittels einer vorab erzeugten globalen Tabelle indiziert; die Klassennamen können dann auch entfernt werden.

- Die übrigen Einträge in den Konstantenpools der Klassen (Strings, Konstanten) werden in einem gemeinsamen Ressourcen-Pool zusammengefasst, der direkt (mit der 16-Bit-Adresse) indiziert wird; das beseitigt die Zwischenschicht bei der Indizierung der Konstantenpool-Einträge und auch die Einschränkung auf 256 Einträge bei kleinen Systemen (siehe Abschnitt 4.2.2).

- Spezielle Methoden, die implizit bzw. über ihren festen Namen gestartet werden (`<clinit>()`, `main(..)`, `finalize()`), bekommen in jeder Klasse reservierte Positionen, die ggf. frei bleiben, wenn diese Methoden nicht existieren. Der Name dieser Methoden wird dann ebenfalls nicht mehr benötigt und die symbolische Suche kann entfallen.

- Optional können die Symbole auch komplett erhalten bleiben, um eine VM zu erzeugen, die Reflection unterstützt.

Fehlende oder nicht vollständige Komponenten

- Unterstützung von `finalize()`- und `classFinalize()`[33]-Methoden bei der Speicherbereinigung.
- Synchrone Klasseninitialisierung (alle Threads müssen vor dem ersten Zugriff auf die Initialisierung einer Klasse warten).
- Generische Array-Klassen zur eindeutigen Typ-Unterscheidung zur Laufzeit. Alternativ kann eine sichere Typ-Bestimmung auch stattfinden, wenn in Array-Objekten die jeweilige Dimensionsstufe abgelegt wird.

Systemerweiterungen

- Unterstützung isolierter Java-Prozesse, die jeweils einen eigenen Satz von statischen Klassenobjekten und Threads verwalten. Die Prozesse beeinflussen sich nicht untereinander und können unabhängig voneinander verwaltet und beendet werden. Das ist eine Kombination aus den im JDK vorhandenen Mechanismen Class-Loader und Thread-Group und kann Erweiterungen aus [C$^+$00, 40] enthalten. Zu beachten ist, dass keine Zustände im nativen Code abgelegt werden dürfen. Das wird mit der in Kapitel 10 vorgesehenen VM-Architektur mit weitestgehend eingeschränktem nativen Code vereinfacht. Isolierte Prozesse sind Voraussetzung für auf Java basierende objektorientierte Betriebssysteme (OOOS).

[33] Diese statische Methode wird beim Entfernen eines Klassenobjekts aufgerufen (als Äquivalent zu `finalize()`, welche beim Entfernen von Instanzen aufgerufen wird).

Anhang E

Messungen und Tests

E.1 Statistik des JCbytecodeWriters nach Bearbeitung von „SystemSetup“

```
[1] aconst_null: 35 (0.4416961)
[2] iconst_m1: 58 (0.73195356)
[3] iconst_0: 282 (3.5588086)
[4] iconst_1: 190 (2.3977787)
[5] iconst_2: 66 (0.8329127)
[6] iconst_3: 70 (0.8833922)
[7] iconst_4: 32 (0.40383646)
[8] iconst_5: 39 (0.49217567)
[16] bipush: 720 (9.08632)
[17] sipush: 98 (1.2367492)
[18] ldc: 233 (2.940434)
[19] ldc_w: 7 (0.088339224)
[21] iload: 413 (5.212014)
[25] aload: 135 (1.7036849)
[26] iload_0: 12 (0.15143867)
[27] iload_1: 70 (0.8833922)
[28] iload_2: 167 (2.1075215)
[29] iload_3: 109 (1.3755679)
[42] aload_0: 677 (8.543665)
[43] aload_1: 99 (1.249369)
[44] aload_2: 35 (0.4416961)
[45] aload_3: 23 (0.29025742)
[46] iaload: 33 (0.41645634)
[50] aaload: 34 (0.42907622)
[51] baload: 24 (0.30287734)
[54] istore: 164 (2.0696619)
[58] astore: 42 (0.5300353)
[59] istore_0: 2 (0.025239779)
[60] istore_1: 15 (0.18929833)
[61] istore_2: 40 (0.50479555)
[62] istore_3: 46 (0.58051485)
[75] astore_0: 1 (0.012619889)
[76] astore_1: 27 (0.34073702)
[77] astore_2: 17 (0.21453811)
[78] astore_3: 14 (0.17667845)
[79] iastore: 28 (0.3533569)
[83] aastore: 175 (2.2084806)
[84] bastore: 4 (0.050479557)
[87] pop: 102 (1.2872287)
[89] dup: 255 (3.2180715)
[92] dup2: 1 (0.012619889)
[96] iadd: 204 (2.5744574)
```

```
[100] isub: 79 (0.99697125)
[104] imul: 66 (0.8329127)
[108] idiv: 80 (1.0095911)
[112] irem: 64 (0.8076729)
[116] ineg: 3 (0.037859667)
[120] ishl: 4 (0.050479557)
[122] ishr: 6 (0.075719334)
[126] iand: 7 (0.088339224)
[130] ixor: 6 (0.075719334)
[132] iinc: 86 (1.0853105)
[146] i2c: 1 (0.012619889)
[153] ifeq: 81 (1.022211)
[154] ifne: 52 (0.65623426)
[155] iflt: 8 (0.100959115)
[156] ifge: 18 (0.22715801)
[157] ifgt: 1 (0.012619889)
[158] ifle: 11 (0.13881877)
[159] if_icmpeq: 40 (0.50479555)
[160] if_icmpne: 41 (0.51741546)
[161] if_icmplt: 51 (0.64361435)
[162] if_icmpge: 34 (0.42907622)
[163] if_icmpgt: 7 (0.088339224)
[164] if_icmple: 26 (0.32811713)
[165] if_acmpeq: 5 (0.063099444)
[166] if_acmpne: 4 (0.050479557)
[167] goto: 229 (2.8899546)
[170] tableswitch: 10 (0.12619889)
[171] lookupswitch: 12 (0.15143867)
[172] ireturn: 64 (0.8076729)
[176] areturn: 26 (0.32811713)
[177] return: 132 (1.6658254)
[178] getstatic: 265 (3.3442707)
[179] putstatic: 60 (0.7571933)
[180] getfield: 457 (5.767289)
[181] putfield: 120 (1.5143867)
[182] invokevirtual: 394 (4.972236)
[183] invokespecial: 106 (1.3377082)
[184] invokestatic: 172 (2.170621)
[185] invokeinterface: 35 (0.4416961)
[187] new: 63 (0.79505306)
[188] newarray: 8 (0.100959115)
[189] anewarray: 47 (0.59313476)
[190] arraylength: 23 (0.29025742)
[191] athrow: 17 (0.21453811)
[192] checkcast: 18 (0.22715801)
[193] instanceof: 7 (0.088339224)
[194] monitorenter: 7 (0.088339224)
[195] monitorexit: 15 (0.18929833)
[198] ifnull: 38 (0.4795558)
[199] ifnonnull: 20 (0.25239778)
Bytecodes: 7924
Type cpool: 1751 (22.097425)
Type cpoolb: 233 (2.940434)
Type index: 840 (10.600707)
Type byte: 1080 (13.629479)
Type int: 22 (0.27763754)
Type branch: 666 (8.404846)
Type wbranch: 0 (0.0)
Type pad: 22 (0.27763754)
Type special: 22 (0.27763754)
```

E.2 Vergleich der Codegröße Bytecode ⇔ ST7

```
Beispielalgorithmus (BenchVMarithm.java, für 16 Bit):
  short y=(short)(i*x+i+i-i/x>>2);

Bytecode (Eclipse-Compiler, Cafebabe):
00 15 iload_1 1B
00 16 iload_2 1C
00 17 imul 68
00 18 iload_1 1B
00 19 iadd 60
00 1A iload_1 1B
00 1B iadd 60
00 1C iload_1 1B
00 1D iload_2 1C
00 1E idiv 6C
00 1F isub 64
00 20 iconst_2 05
00 21 ishr 7A
00 22 int2short 93
00 23 istore_3 3E
------------------------> 15 Bytes

ST7 (Handcodiert, Subroutine für Div vorhanden, Zeropage-Variablen):
                ld      a,_i+1                  ; B
                ld      x,_x+1                  ; D
                push    x
                mul     x,a                     ; x.B*x.D=E.F
                ld      _y+1,a                  ; F sichern
                ld      a,_x                    ; C
                ld      _y,x                    ; E sichern
                ld      x,_i+1                  ; B
                mul     x,a                     ; x.B*C.x=_.E.x
                add     a,_y                    ; E akkumulieren
                ld      _y,a
                ld      a,_y                    ; A
                pop     x                       ; D
                mul     x,a                     ; A.x*x.D=_.E.x
                add     a,_y+1                  ; E akkumulieren
                ld      _y,a
                                                        ; i*x   -> 27 Bytes
                ld      a,_i+1
                add     a,_y+1
                ld      _y+1,a
                ld      a,_i
                adc     a,_y
                ld      _y,a
                                                        ; +i    -> 12 Bytes
                ld      a,_i+1
                add     a,_y+1
                ld      _y+1,a
                ld      a,_i
                adc     a,_y
                ld      _y,a
                                                        ; +i    -> 12 Bytes
                ld      a,_i+1
                ld      _t1+1,a
                ld      a,_i
                ld      _t1,a
                ld      a,_x+1
                ld      _t2+1,a
                ld      a,_x
                ld      _t2,a
```

```
                call    div
                                                        ; i/x   -> 19 Bytes
                ld      a,_y+1
                sub     a,_t+1
                ld      _y+1,a
                ld      a,_y
                sbc     a,_t
                ld      _y,a
                                                        ; -     -> 12 Bytes
                sra     _y
                rrc     _y+1
                sra     _y
                rrc     _y+1
                                                        ; >>    -> 8 Bytes
------------------------> 90 Bytes

=======================> Komprimierung um Faktor 6(!)
```

E.3 Messungen an der Speicherbereinigung

E.3.1 Testprogramme

E.3.1.1 Bibliotheksfunktion: CPUtime

Anmerkung Die Klasse `CPUtime` liest die Daten zur benötigten CPU-Zeit eines Threads oder des System-Treads aus. Zur Erstellung dieses Datensatzes ist eine Erweiterung der VM-Konfiguration (Patch) erforderlich, diese kann mit dem Schalter

```
oThr_CPUmeasure equ     USE             ; [undef|USE] jeder Thread zählt seine Millisekunden
```

in der Datei `Threads.inc` des VM-Kerns aktiviert werden. Da derselbe Speicherbereich für die Messdaten verwendet wird, funktionieren diese Messungen nicht zusammen mit denen des Timers (siehe Anhang E.5).

CPUtime.java

```
/*
 * (c) Copyright 20045 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
 * All Rights Reserved.
 *
 * Created on 15.03.2005
 */

/**
 * CPUtime
 * reads the virtual machine measured values for used CPU time of specified threads.
 * @version 1.0
 * @author boehme
 */
public class CPUtime {

    /**
     * Reads the used CPU time of the current thread.
     * @return CPU time value in ms
```

```
     */
    public static native int getValue();
    /**
     * Reads the used CPU time of the specified thread.
     * @return CPU time in ms
     */
    public static native int getValue(Thread t);

    /**
     * Resets the used CPU time value of the current thread.
     */
    public static native void resetValue();

    /**
     * Resets the used CPU time value of the specified thread.
     */
    public static native void resetValue(Thread t);

    /**
     * Returns the used CPU time of the system thread. In the result array are three values:<ol>
     * <li>mark/sweep phase of the garbage collector</li>
     * <li>compact phase of the garbage collector</li>
     * <li>class loader phase</li></ol>
     * These values are reset to 0.
     * @return three system thread CPU time values in ms
     */
    public static native int[] getSystemValues();
}
```

CPUtime.asm

```
;#############################################################################

;; natbib:         CPUtime

;#############################################################################

;; header:

;; include:

;*****************************************************************************
;; native:         public static native int getValue();
    dc.w            1, 1                ; Stack, Locals
    dc.w            0, l<id>end - l<id>start
;   input:
;   output:         stack               result
;*****************************************************************************
l<id>start:
                    mBc_CallNative
                    move.w  ,pxNat_Temp,,pThr_Current       ; aktuellen Thread benutzen
                    jra     c<class>getValue
l<id>end:

;*****************************************************************************
;; native:         public static native int getValue(java.lang.Thread t);
    dc.w            1, 1                ; Stack, Locals
    dc.w            0, l<id>end - l<id>start
;   input:          0                   thread (ungeprüft)
;   output:         stack               result
;*****************************************************************************
l<id>start:
                    mBc_CallNative
                    mFr_getLcVars
                    mFr_rdLcVar.w ,pxNat_Temp,0             ; diesen Thread benutzen
```

```
35                  mHp_ResHandle pxNat_Temp,pxNat_Temp
    c<class>getValue:
        ifdef       oThr_CPUmeasure
                    move.w  ,vNat_Temp1,iThr_CPUtime,pxNat_Temp ; CPUtime auslesen
                    mcp.w   ,pxNat_Temp,,pThr_Current
40                  jrne    l<id>NotSelf
                    madd.w  ,vNat_Temp1,,vThr_Millis        ; beim aktuellen Thread Wert korrigieren
    l<id>NotSelf:
        endif
                    mFr_PushOps.w ,vNat_Temp1
45                  mBc_LeaveNative
                    mNat_IReturn
    l<id>end:

    ;***************************************************************************
50  ;; native:      public static native void resetValue();
        dc.w        0, 1                ; Stack, Locals
        dc.w        0, l<id>end - l<id>start
    ;   input:
    ;   output:
55  ;***************************************************************************
    l<id>start:
                    mBc_CallNative
                    move.w  ,pxNat_Temp,,pThr_Current       ; aktuellen Thread benutzen
                    jra     c<class>resetValue
60  l<id>end:

    ;***************************************************************************
    ;; native:      public static native void resetValue(java.lang.Thread t);
        dc.w        0, 1                ; Stack, Locals
65      dc.w        0, l<id>end - l<id>start
    ;   input:      0                   thread (ungeprüft)
    ;   output:
    ;***************************************************************************
    l<id>start:
70                  mBc_CallNative
                    mFr_getLcVars
                    mFr_rdLcVar.w ,pxNat_Temp,0             ; diesen Thread benutzen
                    mHp_ResHandle pxNat_Temp,pxNat_Temp
    c<class>resetValue:
75      ifdef       oThr_CPUmeasure
                    move.w  ,vNat_Temp1,#,0                 ; CPUtime rücksetzen
                    mcp.w   ,pxNat_Temp,,pThr_Current
                    jrne    l<id>NotSelf
                    msub.w  ,vNat_Temp1,,vThr_Millis        ; beim aktuellen Thread Wert korrigieren
80  l<id>NotSelf:
                    move.w  iThr_CPUtime,pxNat_Temp,,vNat_Temp1 ; CPUtime schreiben
        endif
                    mBc_LeaveNative
                    mNat_Return
85  l<id>end:

    ;***************************************************************************
    ;; native:      public static native int[] getSystemValues();
        dc.w        1, 0                ; Stack, Locals
90      dc.w        0, l<id>end - l<id>start
    ;   input:
    ;   output:     stack               result in int[]
    ;***************************************************************************
    l<id>start:
95  <code>
                dc.b    $06     ; iconst_3
                dc.b    $bc,$0a ; newarray int
    l<id>loop:
                mBc_CallNative
```

```
100                 mbtjt   ST_Sleeping,l<id>cont       ; GC muss idle sein
                    mBc_LeaveNative yield
                    mBc_Goto _Bc_GOTO,l<id>loop
    l<id>cont:
                    move    ,pHp_Temp,y,pFr_opStackH
105                 move    ,pHp_Temp+1,y,pFr_opStackL  ; TOP
                    mBc_GetArraybytes pHp_Temp,pHp_Temp ; Ziel
                    move.w  ,pHp_Temp2,#,$140           ; Quelle
                    move.w  ,pHp_Temp3,#,6              ; Anz.
                    call    cHp_MemCopy                 ; kopieren
110                 move.w  ,pHp_Temp2,#,$140           ; Adresse
                    move.w  ,pHp_Temp,#,6               ; Anz.
                    call    cHp_DirectClear             ; initialisieren
                    mBc_LeaveNative
                    mNat_AReturn
115 l<id>end:
```

E.3.1.2 Testprogramm: GCanalyser

GCanalyzer.java

```
    /*
     * (c) Copyright 2005 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
     * All Rights Reserved.
     *
  5  * Created on 15.03.2005
     */

    import java.io.IOException;

 10 import jcontrol.comm.ConsoleOutputStream;
    import jcontrol.comm.RS232;
    import jcontrol.lang.Deadline;
    import jcontrol.lang.DeadlineMissException;
    import jcontrol.system.Management;
 15
    /**
     * GCanalyzer performs some CPU time analysis of the Garbage Collector.
     * @version 1.0
     * @author boehme
 20  */
    public class GCanalyzer {

        /** All program outputs use this stream */
        static ConsoleOutputStream con;
 25
        /**
         * Program entry point.
         * @param args not used
         */
 30     public static void main(String[] args) {
            try {
                con=new RS232();
                int depth=3;
                int[] run={500,250,100,50,0};
 35             con.print("JControl Garbage Collector CPU Usage Analysis (depth=");
                con.print(String.valueOf(depth));
                con.println(")");
                for(int r=0;r<run.length;r++){
                    con.println("Delay, Anz, Iterations, Real, User, Mark, Sweep");
 40                 for(int i=0;i<18;i++){
                        test(i,depth,run[r]);
```

```
                }
            }
            Management.halt();
        } catch(IOException e) {}

    }

    /**
     * Performs a single parametized test.
     * @param width number of trees
     * @param depth of tree
     * @param rate for allocations
     */
    static void test(int width, int depth, int rate){
        con.print(String.valueOf(rate));
        con.print(", ");
        GCanalyzer[] gcas=new GCanalyzer[width];
        count=0;
        for(int i=0;i<width;i++){
            gcas[i]=build(depth);
        }
        con.print(String.valueOf(count));
        boolean over=false;
        CPUtime.getSystemValues(); // reset
        CPUtime.resetValue();
        int realTime=Management.currentTimeMillis();
        int count=0;
        if(rate>0){
            Deadline d=new Deadline(rate);
            int i=0;
            int max=5000/rate;
            for(count=0;count<max;count++) {
                try {
                    synchronized(d) {
                        for(;count<max;count++) {
                            if(width>0) {
                                gcas[i]=null;
                                gcas[i]=build(depth);
                                i=(i+1)%width;
                            }
                            d.append(rate);
                        }
                    }
                } catch(DeadlineMissException e) {
                    over=true;
                    if((Management.currentTimeMillis()-realTime)>5000) break;
                }
            }
        } else {
            int i=0;
            for(count=0;;count++) {
                if(width>0) {
                    gcas[i]=null;
                    gcas[i]=build(depth);
                    i=(i+1)%width;
                }
                if((Management.currentTimeMillis()-realTime)>5000) break;
            }
        }
        gcas=null;
        realTime=Management.currentTimeMillis()-realTime;
        int[] values=CPUtime.getSystemValues(); // read
        if(over) con.print("*");
        con.print(", ");
        con.print(String.valueOf(count-1));
```

```
            con.print(", ");
            con.print(String.valueOf(realTime));
            con.print(", ");
            con.print(String.valueOf(CPUtime.getValue()));
            con.print(", ");
            con.print(String.valueOf(values[0]));
            con.print(", ");
            con.print(String.valueOf(values[1]));
            con.println();
        }

        /** Tree depth counter */
        static int count;

        /**
         * Creates a binary tree of specified depth. The number of created objects is
         * 1, 3, 7, 15, 31, ...
         * @param depth to use
         * @return binary tree of GCanalyzer
         */
        static GCanalyzer build(int depth) {
            if(depth<=0) return null;
            count++;
            return new GCanalyzer(build(depth-1),build(depth-1));
        }

        /** Used to build a binary tree */
        GCanalyzer left,right;

        /**
         * Constructs an new GCanalyzer tree leaf with no childs.
         */
        GCanalyzer() {
            left=null;
            right=null;
        }

        /**
         * Constructs a new GCanalyzer tree node with specified childs.
         * @param useLeft child
         * @param useRight child
         */
        GCanalyzer(GCanalyzer useLeft, GCanalyzer useRight) {
            left=useLeft;
            right=useRight;
        }

    }
```

E.3.2 Testausgabe

E.3.2.1 1. Durchlauf mit Verwendung des einfachen Kompaktierers mit sequentieller Blocksuche

GCanalysis_alt.csv

```
Delay,Anz,Iterations,Real,User,Mark,Sweep
500,0,10,5519,28,10,18
500,7,10,5541,208,137,260
500,14,10,5520,214,143,376
```

```
500,21,10,5522,213,176,554
500,28,10,5524,212,201,763
500,35,10,5526,216,229,1010
500,42,10,5527,218,251,1306
500,49,10,5529,220,277,1616
500,56,10,5531,220,303,1968
500,63,10,5532,222,333,2359
500,70,10,5534,223,362,2776
500,77,10,5537,226,387,3228
500,84,10,5537,226,421,3736
500,91,10,5539,231,452,4809
500,98,10,5540,231,433,4920
500,105,10,5638,232,483,5341
500,112,10,5504,226,421,5373
500,119,10,5502,226,368,5210
Delay,Anz,Iterations,Real,User,Mark,Sweep
250,0,20,5252,35,13,27
250,7,20,5277,399,315,792
250,14,20,5278,401,350,1110
250,21,20,5278,404,385,1470
250,28,20,5279,405,423,1874
250,35,20,5279,408,467,2331
250,42,20,5280,410,498,2798
250,49,20,5281,411,537,3310
250,56,20,5281,407,574,4829
250,63,20,5282,412,611,4797
250,70,20,5284,414,583,4888
250,77,20,5286,418,475,5057
250,84,20,5334,423,989,4678
250,91,20,5275,432,476,5181
250,98,20,5369,435,496,5258
250,105,20,5256,429,490,5401
250,112,20,5254,428,424,5396
250,119,20,5254,438,418,5357
Delay,Anz,Iterations,Real,User,Mark,Sweep
100,0,50,5102,17,13,29
100,7,50,5128,949,740,1875
100,14,50,5129,963,917,2894
100,21,50,5130,957,1026,4885
100,28,50,5130,957,862,5021
100,35,50,5130,965,1458,4400
100,42,50,5176,973,684,5321
100,49,50,5132,977,681,5329
100,56,50,5127,984,589,5435
100,63,50,5122,1002,695,5266
100,70,50,5126,1018,622,5400
100,77,50,5241,1034,728,5382
100,84,50,5103,1036,490,5677
100,91*,35,5168,841,542,5217
100,98*,30,5345,734,546,5521
100,105*,22,5239,564,333,5373
100,112*,16,5372,462,345,5643
100,119*,9,5004,288,313,5102
Delay,Anz,Iterations,Real,User,Mark,Sweep
50,0,100,5052,117,14,34
50,7,100,5078,1851,2629,3659
50,14,100,5077,1857,1228,5299
50,21,100,5078,1879,2005,4667
50,28,100,5072,1911,1320,5449
50,35,100,5105,1933,1032,5773
50,42,100,5082,1960,789,6205
50,49,100,5141,2024,757,6142
50,56*,74,5073,1611,592,5737
50,63*,74,5668,1634,600,6135
50,70*,57,5246,1298,353,5483
```

```
50,77*,51,5294,1167,290,5216
50,84*,42,5305,967,331,5834
50,91*,35,5181,830,308,5561
50,98*,29,5271,701,313,5593
50,105*,23,5396,599,325,5682
50,112*,16,5373,440,330,5715
50,119*,9,5031,281,313,5157
Delay,Anz,Iterations,Real,User,Mark,Sweep
0,0,8544,5002,5018,0,0
0,7,222,5015,4383,297,373
0,14,215,5013,4320,304,426
0,21,205,5014,4168,312,580
0,28,182,5006,3781,295,2068
0,35,174,5006,3684,286,2291
0,42,164,5006,3369,294,2385
0,49,146,5037,3157,292,2723
0,56,139,5102,3024,301,2688
0,63,133,5123,2913,312,2651
0,70,122,5015,2701,319,2359
0,77,113,5008,2414,329,2594
0,84,100,5018,2170,336,2798
0,91,89,5088,1963,343,3028
0,98,78,5115,1726,345,3257
0,105,67,5081,1464,346,3419
0,112,56,5163,1232,353,3706
0,119,45,5205,994,356,3923
```

E.3.2.2 2. Durchlauf mit Verwendung des aufwändigeren Kompaktieres mit verketteten Listen

GCanalysis_neu.csv

```
Delay,Anz,Iterations,Real,User,Mark,Sweep
500,0,10,5514,29,11,3
500,7,10,5516,212,128,31
500,14,10,5516,216,145,44
500,21,10,5518,215,174,56
500,28,10,5519,216,197,73
500,35,10,5520,217,227,84
500,42,10,5522,220,252,101
500,49,10,5523,221,279,109
500,56,10,5524,222,304,125
500,63,10,5524,227,332,138
500,70,10,5527,225,360,149
500,77,10,5529,228,384,164
500,84,10,5530,230,419,181
500,91,10,5531,237,448,191
500,98,10,5532,235,476,203
500,105,10,5533,233,492,213
500,112,10,5535,235,517,228
500,119,10,5536,235,543,242
Delay,Anz,Iterations,Real,User,Mark,Sweep
250,0,20,5264,39,11,3
250,7,20,5278,394,227,68
250,14,20,5266,404,265,81
250,21,20,5267,407,324,107
250,28,20,5269,403,364,137
250,35,20,5270,407,424,160
250,42,20,5271,414,467,194
250,49,20,5273,414,521,214
250,56,20,5274,415,573,240
250,63,20,5275,420,623,266
```

```
250,70,20,5277,425,677,291
250,77,20,5278,423,724,319
250,84,20,5279,428,787,346
250,91,20,5281,435,843,375
250,98,20,5282,440,897,397
250,105,20,5283,439,934,421
250,112,20,5284,439,987,442
250,119,20,5285,445,1036,473
Delay,Anz,Iterations,Real,User,Mark,Sweep
100,0,50,5102,18,13,2
100,7,50,5119,974,729,117
100,14,50,5120,968,867,187
100,21,50,5121,969,958,256
100,28,50,5121,979,1054,332
100,35,50,5122,987,1150,391
100,42,50,5123,993,1255,458
100,49,50,5124,993,1355,522
100,56,50,5125,998,1440,583
100,63,50,5126,1008,1538,652
100,70,50,5127,1011,1638,714
100,77,50,5128,1017,1743,787
100,84,50,5130,1025,1882,856
100,91,50,5130,1047,2003,921
100,98,50,5132,1048,2132,975
100,105,50,5133,1050,2240,1036
100,112,50,5134,1050,2344,1091
100,119,50,5136,1055,2457,1162
Delay,Anz,Iterations,Real,User,Mark,Sweep
50,0,100,5052,30,13,1
50,7,100,5069,1925,1409,235
50,14,100,5070,1905,1703,369
50,21,100,5071,1907,1881,511
50,28,100,5072,1927,2428,679
50,35,100,5071,1919,4570,801
50,42,100,5073,1910,4830,1010
50,49,100,5074,1918,4409,1978
50,56,100,5075,1933,3574,2801
50,63,100,5076,1951,2883,3519
50,70,100,5077,1968,5068,1624
50,77,100,5078,1986,3625,2873
50,84,100,5070,2002,3545,3003
50,91,100,5080,2023,3613,3084
50,98,100,5083,2030,3653,2933
50,105,100,5083,2043,3938,2905
50,112*,100,5874,2168,3377,1850
50,119*,52,5196,1328,1839,1005
Delay,Anz,Iterations,Real,User,Mark,Sweep
0,0,8543,5001,5017,0,0
0,7,235,5015,4658,257,121
0,14,233,5012,4647,258,133
0,21,229,5005,4583,283,155
0,28,226,5008,4551,303,172
0,35,222,5009,4511,328,188
0,42,218,5006,4468,347,210
0,49,214,5016,4430,368,231
0,56,210,5007,4401,385,246
0,63,207,5021,4352,412,268
0,70,201,5018,4275,458,301
0,77,200,5061,4179,525,355
0,84,192,5008,4054,570,390
0,91,184,5016,3902,650,460
0,98,178,5011,3734,737,525
0,105,167,5004,3512,849,616
0,112,154,5006,3239,986,740
0,119,137,5014,2887,1172,898
```

E.4 Benchmarks

E.4.1 Testprogramme

E.4.1.1 Basis der Messungen: BenchVM

BenchVM.java

```
/*
 * (c) Copyright 1998-2005 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
 * All Rights Reserved.
 */
package jcontrol.benchmark;

/**
 * BenchVM, comparison base for all measurements (loop with dummy body).
 * @author boehme
 */
public class BenchVM{

  /**
   * Program entry point.
   * @param args is partially used for return values:<ol>
   * <li>returns the number of processed bytecodes</li>
   * <li>returns the number of processed operations</li>
   * <li>takes the number of loops (defaults to 10000)</li>
   * </ol>
   */
  public static void main(String args[]){
    int loops=10000;
    if(args!=null){
        args[0]="0"; args[1]="0";
        if(args.length>2) loops=Integer.parseInt(args[2]);
    }
    for(int i=0;i<loops;i++){
      int x=i;
      x++;
    }
  }

  /**
   * Used for testing <code>invokestatic</code> and <code>return</code>.
   * @param var dummy parameter
   */
  static void smethod(int var){}

  /**
   * Used for testing <code>invokevirtual</code> and <code>return</code>.
   */
  void imethod(){}

  /** Used for testing <code>putfield</code>. */
  int ivar;

  /** Used for testing <code>putstatic</code>. */
  static int svar;
}
```

E.4.1.2 Messprogramme

BenchVMarithm.java

```
/*
 * (c) Copyright 1998-2005 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
 * All Rights Reserved.
 */
package jcontrol.benchmark;

/**
 * BenchVMarithm performs some arithmetic operations, takes 14 bytecodes or 6 operations per loop.
 * @author boehme
 */
public class BenchVMarithm{

  /**
   * @see BenchVM
   */
  public static void main(String args[]){
    int loops=10000;
    if(args!=null){
        args[0]="14"; args[1]="6";
        if(args.length>2) loops=Integer.parseInt(args[2]);
    }
    loops++;
    for(int i=1;i<loops;i++){
      int x=i;
      int y=i*x+i+i-i/x>>2;
      y++;
    }
  }

  static void smethod(int var){}
  void imethod(){}
  int ivar;
  static int svar;
}
```

BenchVMcond.java

```
/*
 * (c) Copyright 1998-2005 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
 * All Rights Reserved.
 */
package jcontrol.benchmark;

/**
 * BenchVMcond performs some conditional operations, takes 4 bytecodes or 1 operation per loop.
 * @author boehme
 */
public class BenchVMcond{

  /**
   * @see BenchVM
   */
  static void main(String args[]){
    int loops=10000;
    if(args!=null){
        args[0]="4"; args[1]="1";
        if(args.length>2) loops=Integer.parseInt(args[2]);
    }
    int loops2=loops/2;
    for(int i=0;i<loops;i++){
      int x=i;
```

```
      if(x>loops2)
        x++;
      else
        x++;
    }
  }

  static void smethod(int var){}
  void imethod(){}
  int ivar;
  static int svar;
}
```

BenchVMarray.java

```
/*
 * (c) Copyright 1998-2005 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
 * All Rights Reserved.
 */
package jcontrol.benchmark;

/**
 * BenchVMarray performs some array storage, takes 4 bytecodes or 1 operation per loop.
 * @author boehme
 */
public class BenchVMarray{

  /**
   * @see BenchVM
   */
  public static void main(String args[]){
    int loops=10000;
    if(args!=null){
        args[0]="4"; args[1]="1";
        if(args.length>2) loops=Integer.parseInt(args[2]);
    }
    int a[]=new int[100];
    for(int i=0;i<loops;i++){
      int x=i;
      x++;
      a[1]=x;
    }
  }

  static void smethod(int var){}
  void imethod(){}
  int ivar;
  static int svar;
}
```

BenchVMsvar.java

```
/*
 * (c) Copyright 1998-2005 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
 * All Rights Reserved.
 */
package jcontrol.benchmark;

/**
 * BenchVMsvar performs some static variable accesses, takes 4 bytecodes or 2 operations per loop.
 * @author boehme
 */
public class BenchVMsvar{

  /**
```

```
     * @see BenchVM
    */
   public static void main(String args[]){
     int loops=10000;
     if(args!=null){
         args[0]="4"; args[1]="2";
         if(args.length>2) loops=Integer.parseInt(args[2]);
     }
     for(int i=0;i<loops;i++){
       int x=i;
       svar=x;
       x++;
       x=svar;
     }
   }

   static void smethod(int var){}
   void imethod(){}
   int ivar;
   static int svar;
}
```

BenchVMivar.java

```
/*
 * (c) Copyright 1998-2005 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
 * All Rights Reserved.
 */
package jcontrol.benchmark;

/**
 * BenchVMivar performs some instance variable accesses, takes 6 bytecodes or 2 operations per loop.
 * @author boehme
 */
public class BenchVMivar{

  /**
   * @see BenchVM
   */
  public static void main(String args[]){
    int loops=10000;
    if(args!=null){
        args[0]="6"; args[1]="2";
        if(args.length>2) loops=Integer.parseInt(args[2]);
    }
    BenchVMivar obj=new BenchVMivar();
    for(int i=0;i<loops;i++){
      int x=i;
      obj.ivar=x;
      x++;
      x=obj.ivar;
    }
  }

  static void smethod(int var){}
  void imethod(){}
  int ivar;
  static int svar;
}
```

BenchVMsmet.java

```
/*
 * (c) Copyright 1998-2005 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
 * All Rights Reserved.
```

```
 */
package jcontrol.benchmark;

/**
 * BenchVMsmet performs some static method calls (using 1 parameter), takes 3 bytecodes or
 * 1 operation per loop.
 * @author boehme
 */
public class BenchVMsmet{

  /**
   * @see BenchVM
   */
  public static void main(String args[]){
    int loops=10000;
    if(args!=null){
        args[0]="3"; args[1]="1";
        if(args.length>2) loops=Integer.parseInt(args[2]);
    }
    for(int i=0;i<loops;i++){
      smethod(i);
    }
  }

  static void smethod(int var){}
  void imethod(){}
  int ivar;
  static int svar;
}
```

BenchVMimet.java

```
/*
 * (c) Copyright 1998-2005 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
 * All Rights Reserved.
 */
package jcontrol.benchmark;

/**
 * BenchVMimet performs some virtual method calls, takes 3 bytecodes or 1 operation per loop.
 * @author boehme
 */
public class BenchVMimet{

  /**
   * @see BenchVM
   */
  public static void main(String args[]){
    int loops=10000;
    if(args!=null){
        args[0]="3"; args[1]="1";
        if(args.length>2) loops=Integer.parseInt(args[2]);
    }
    BenchVMimet obj=new BenchVMimet();
    for(int i=0;i<loops;i++){
      int x=i;
      x++;
      obj.imethod();
    }
  }

  static void smethod(int var){}
  void imethod(){}
  int ivar;
  static int svar;
```

```
}
```

BenchVMobj.java

```
/*
 * (c) Copyright 1998-2005 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
 * All Rights Reserved.
 */
package jcontrol.benchmark;

/**
 * BenchVMobj performs some object creations (implies 2 method calls), takes 6 bytecodes or
 * 3 operations per loop.
 * @author boehme
 */
public class BenchVMobj{

  /**
   * @see BenchVM
   */
  public static void main(String args[]){
    int loops=10000;
    if(args!=null){
        args[0]="6"; args[1]="3";
        if(args.length>2) loops=Integer.parseInt(args[2]);
    }
    for(int i=0;i<loops;i++){
      int x=i;
      x++;
      new BenchVMobj();
    }
  }

  static void smethod(int var){}
  void imethod(){}
  int ivar;
  static int svar;
}
```

BenchVMobj1.java

```
/*
 * (c) Copyright 1998-2005 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
 * All Rights Reserved.
 */
package jcontrol.benchmark;

/**
 * BenchVMobj1 performs some object creations (for measurements without implicit method calls),
 * takes 1 bytecode or 1 operation per loop.
 * @author boehme
 */
public class BenchVMobj1{

  /**
   * @see BenchVM
   */
  public static void main(String args[]){
    int loops=10000;
    if(args!=null){
        args[0]="1"; args[1]="1";
        if(args.length>2) loops=Integer.parseInt(args[2]);
    }
    for(int i=0;i<loops;i++){
      int x=i;
```

```
25        x++;
          new BenchVMobj1();
        }
      }

30    static void smethod(int var){}
      void imethod(){}
      int ivar;
      static int svar;
    }
```

E.4.1.3 Hauptprogramme zum Start der Tests auf verschiedenen Plattformen

Die oben aufgeführten Testprogramme sind so realisiert, dass sie mit unterschiedlichen Programmierschnittstellen ausgeführt werden können (quelltextkompatibel; sie müssen also für die unterschiedlichen Zielplattformen gegen das entsprechende API übersetzt werden). Die von den Programmierschnittstellen abhängigen Komponenten der Tests wurden zu entsprechenden Startern zusammengefasst, die für die jeweiligen Zielplattform ausgelegt sind. Dabei verwendet die Variante für die 8-Bit-Version des *JControl*-APIs den Mechanismus, in den Ressourcen nach Klassen zu suchen und diese der Reihe nach aufzurufen. Für die Messung der threadweise verwendeten Rechenzeit wird die Routine `CPUtime` aus Anhang E.3.1.1 benutzt. Die Version für das JDK (J2SE) kann den Reflection-Mechanismus verwenden; die übrigen Realisierungen müssen die Klassen direkt aufrufen.

TestAll.java

```
    /*
     * (c) Copyright 1998-2005 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
     * All Rights Reserved.
     */
  5 package jcontrol.benchmark;

    import jcontrol.io.Resource;
    import jcontrol.lang.ThreadExt;
    import jcontrol.system.CPUtime;
 10 import jcontrol.system.Management;

    /**
     * TestAll, performs all benchmarks using the JControl API.
     * @author boehme
 15  */
    public class TestAll {

        /**
         * Program entry point.<br>
 20      * Performs all tests found in the resources using the name pattern <code>BenchVM</code>.
         * @param args not used
         */
        public static void main(String args[]){
          try{
 25 //         System.setOut(new DisplayConsole());   // optionally output on display
            System.out.println("Jcontrol Benchmarks");
```

```
        System.out.println("Test,Real,User,BCs,OPs,BCs/s,OPs/s,cy/BC,cy/OP");
        String name,sub=null;
        Resource rom=new Resource(Resource.FLASHACCESS);
        do {
            name=rom.getName();
            int pos=name.indexOf("BenchVM",0);
            if(pos>-1) {
                sub=name.substring(pos,name.length());
                if(sub.equals("BenchVM")) break;
            }
            rom=rom.next();
        } while(rom!=null);
        System.out.print(sub.concat("(base),"));
        args=new String[2];
        int time=Management.currentTimeMillis();
        CPUtime.resetValue();
        Management.start(name,args,false);
        int base=Management.currentTimeMillis()-time;
        int ubase=CPUtime.getValue();
        System.out.print(String.valueOf(base).concat(","));
        System.out.println(String.valueOf(ubase).concat(",,,,,,"));

        rom=new Resource(Resource.FLASHACCESS);
        do {
            name=rom.getName();
            int pos=name.indexOf("BenchVM",0);
            if(pos>-1) {
                sub=name.substring(pos,name.length());
                if(!sub.equals("BenchVM")) {
                    System.out.print(sub.concat(","));
                    time=Management.currentTimeMillis();
                    CPUtime.resetValue();
                    Management.start(name,args,false);
                    int udiff=CPUtime.getValue()-ubase;
                    int diff=Management.currentTimeMillis()-time-base;
                    System.out.print(String.valueOf(diff).concat(","));
                    System.out.print(String.valueOf(udiff).concat(","));
                    System.out.print(args[0].concat(","));
                    System.out.print(args[1].concat(","));
                    int bcs=Integer.parseInt(args[0]);
                    int ops=Integer.parseInt(args[1]);
                    int ps=bcs*1000/(udiff/10);
                    int psr=(bcs*1000%(udiff/10))*10/(udiff/1000);
                    if(psr>1000) {ps++; psr-=1000;}
                    String s=String.valueOf(psr); while(s.length()<3) s="0".concat(s);
                    System.out.print(String.valueOf(ps));
                    System.out.print(s.concat(","));
                    ps=ops*1000/(udiff/10);
                    psr=(ops*1000%(udiff/10))*10/(udiff/1000);
                    if(psr>1000) {ps++; psr-=1000;}
                    s=String.valueOf(psr); while(s.length()<3) s="0".concat(s);
                    System.out.print(String.valueOf(ps));
                    System.out.print(s.concat(","));
                    System.out.print(String.valueOf(udiff/5*4/bcs).concat(","));
                    System.out.println(String.valueOf(udiff/5*4/ops));
                }
            }
            rom=rom.next();
        } while(rom!=null);
        System.out.close();
        try {
            ThreadExt.sleep(25000);
        } catch (InterruptedException e) {}
      }catch(java.io.IOException e){
             System.out.println("IOException");
```

```
      }
    }
}
```

TestAllOnJCVM32.java

```
/*
 * (c) Copyright 2005 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
 * All Rights Reserved.
 *
 * Created on 24.06.2005
 */
package jcontrol.benchmark;

/**
 * TestAllOnJCVM32, performs all benchmarks using the JControl API on the JCVM32.
 * @author boehme
 */
public class TestAllOnJCVM32 {

    static final int LOOPS=10000;
    static final int CLOCK=54;

    private static final int loops=LOOPS*CLOCK/8;
    private static String[] benchArgs=new String[3];;
    private static long time;

    /**
     * Initializes test.
     * @param name of test
     */
    static void starttest(String name){
        System.out.print(name);
        time=(int)System.currentTimeMillis();
        benchArgs[2]=String.valueOf(loops);
    }

    /**
     * Test analysis and printout.
     * @param base timebase for test (usually from BenchVM)
     * @return test duration
     */
    static int test(int base){
        time=(int)System.currentTimeMillis()-time;
        int duration=(int)time-base;
        System.out.print(","+duration);
        System.out.print(","+benchArgs[0]);
        System.out.print(","+benchArgs[1]);
        int bcs=Integer.parseInt(benchArgs[0]);
        int ops=Integer.parseInt(benchArgs[1]);
        int bcsps=bcs*loops*1000/duration;
        System.out.print(","+bcsps);
        int opsps=ops*loops*1000/duration;
        System.out.print(","+opsps+",");
        if(bcsps>0) System.out.print(CLOCK*1000000/bcsps);
        System.out.print(",");
        if(opsps>0) System.out.print(CLOCK*1000000/opsps);
        System.out.println();
        return duration;
    }

    /**
     * Program entry point.
     * Explicetly performs all tests.
     * @param args not used
```

```
60      */
        public static void main(String[] args) {
            System.out.println("Jcontrol Benchmarks (JCVM32, "+LOOPS*CLOCK/8+" loops)");
            System.out.println("Test,Real,BCs,OPs,BCs/s,OPs/s,cy/BC,cy/OP");
            starttest("BenchVM");
65          BenchVM.main(benchArgs);
            int base=test(0);
            starttest("BenchVMarithm");
            BenchVMarithm.main(benchArgs);
            test(base);
70          starttest("BenchVMcond");
            BenchVMcond.main(benchArgs);
            test(base);
            starttest("BenchVMarray");
            BenchVMarray.main(benchArgs);
75          test(base);
            starttest("BenchVMsvar");
            BenchVMsvar.main(benchArgs);
            test(base);
            starttest("BenchVMivar");
80          BenchVMivar.main(benchArgs);
            test(base);
            starttest("BenchVMsmet");
            BenchVMsmet.main(benchArgs);
            test(base);
85          starttest("BenchVMimet");
            BenchVMimet.main(benchArgs);
            int imet=test(base);
            starttest("BenchVMobj");
            BenchVMobj.main(benchArgs);
90          test(base);
            starttest("BenchVMobj1");
            BenchVMobj1.main(benchArgs);
            test(base+2*imet);
        }
95
    }
```

TestAllOnJ2ME.java

```
    /*
     * (c) Copyright 2005 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
     * All Rights Reserved.
     *
 5   * Created on 24.06.2005
     */
    package jcontrol.benchmark;

    /**
10   * TestAllOnJ2ME, performs all benchmarks using the J2ME on KVM.
     * @author boehme
     */
    public class TestAllOnJ2ME {

15      static final int LOOPS=100000;
        static final int CLOCK=1600;

        private static final int loops=LOOPS*CLOCK/8;
        private static String[] benchArgs=new String[3];;
20      private static long time;

        /**
         * Initializes test.
         * @param name of test
25       */
```

```
    static void starttest(String name){
        System.out.print(name);
        time=(int)System.currentTimeMillis();
        benchArgs[2]=String.valueOf(loops);
    }

    /**
     * Test analysis and printout.
     * @param base timebase for test (usually from BenchVM)
     * @return test duration
     */
    static int test(int base){
        time=(int)System.currentTimeMillis()-time;
        int duration=(int)time-base;
        System.out.print(","+duration);
        System.out.print(","+benchArgs[0]);
        System.out.print(","+benchArgs[1]);
        long bcs=Long.parseLong(benchArgs[0]);
        long ops=Long.parseLong(benchArgs[1]);
        long bcsps=bcs*loops*1000/duration;
        System.out.print(","+bcsps);
        long opsps=ops*loops*1000/duration;
        System.out.print(","+opsps+",");
        if(bcsps>0) System.out.print((long)CLOCK*1000000/bcsps);
        System.out.print(",");
        if(opsps>0) System.out.print((long)CLOCK*1000000/opsps);
        System.out.println();
        return duration;
    }

    /**
     * Program entry point.
     * Explicetly performs all tests.
     * @param args not used
     */
    public static void main(String[] args) {
        System.out.println("Jcontrol Benchmarks (J2ME KVM, "+LOOPS*CLOCK/8+" loops)");
        System.out.println("Test,Real,BCs,OPs,BCs/s,OPs/s,cy/BC,cy/OP");
        starttest("BenchVM");
        BenchVM.main(benchArgs);
        int base=test(0);
        starttest("BenchVMarithm");
        BenchVMarithm.main(benchArgs);
        test(base);
        starttest("BenchVMcond");
        BenchVMcond.main(benchArgs);
        test(base);
        starttest("BenchVMarray");
        BenchVMarray.main(benchArgs);
        test(base);
        starttest("BenchVMsvar");
        BenchVMsvar.main(benchArgs);
        test(base);
        starttest("BenchVMivar");
        BenchVMivar.main(benchArgs);
        test(base);
        starttest("BenchVMsmet");
        BenchVMsmet.main(benchArgs);
        test(base);
        starttest("BenchVMimet");
        BenchVMimet.main(benchArgs);
        int imet=test(base);
        starttest("BenchVMobj");
        BenchVMobj.main(benchArgs);
        test(base);
```

```
        starttest("BenchVMobj1");
        BenchVMobj1.main(benchArgs);
        test(base+2*imet);
    }

}
```

TestAllOnJDK.java

```
/*
 * (c) Copyright 2005 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
 * All Rights Reserved.
 *
 * Created on 24.06.2005
 */
package jcontrol.benchmark;

import java.lang.reflect.Method;

/**
 * TestAllOnJDK, performs all benchmarks using the J2SE JDK on Hotspot.
 * @author boehme
 */
public class TestAllOnJDK {

    static final int LOOPS=100000;
    static final int CLOCK=1600;

    /**
     * Performs a single Test known by name (using reflection) and prints the results.
     * @param name of test
     * @param base timebase for test (usually from BenchVM)
     * @return test duration
     */
    static int test(String name, int base){
        String[] args=new String[3];
        int loops=LOOPS*CLOCK/8;
        args[2]=String.valueOf(loops);
        int duration=0;
        try {
            Class test=Class.forName("jcontrol.benchmark."+name);
            Method m=test.getDeclaredMethod("main",new Class[] {args.getClass()});
            System.out.print(name);
            long time=System.nanoTime(); // JDK1.5 only
            m.invoke(null,new Object[]{args});
            time=(System.nanoTime()-time)/1000000; // JDK1.5 only
            duration=(int)time-base;
            System.out.print(","+duration);
            System.out.print(","+args[0]);
            System.out.print(","+args[1]);
            long bcs=Long.parseLong(args[0]);
            long ops=Long.parseLong(args[1]);
            long bcsps=bcs*loops*1000/duration;
            System.out.print(","+bcsps);
            long opsps=ops*loops*1000/duration;
            System.out.print(","+opsps+",");
            if(bcsps>0) System.out.print((long)CLOCK*1000000/bcsps);
            System.out.print(",");
            if(opsps>0) System.out.print((long)CLOCK*1000000/opsps);
            System.out.println();
        } catch(Exception e) {
            e.printStackTrace();
        }
        return duration;
    }
```

```
    /**
     * Program entry point.
     * Explicetly performs all tests.
     * @param args not used
     */
    public static void main(String[] args) {
        System.out.println("Jcontrol Benchmarks ("+System.getProperty("java.vm.version")+", "+
                System.getProperty("os.arch")+", "+System.getProperty("os.name")+", "+LOOPS*CLOCK/8+" loops)");
        System.out.println("Test,Real,BCs,OPs,BCs/s,OPs/s,cy/BC,cy/OP");
        int base=test("BenchVM",0);
        test("BenchVMarithm",base);
        test("BenchVMcond",base);
        test("BenchVMarray",base);
        test("BenchVMsvar",base);
        test("BenchVMivar",base);
        test("BenchVMsmet",base);
        int imet=test("BenchVMimet",base);
        test("BenchVMobj",base);
        test("BenchVMobj1",base+2*imet);
    }

}
```

E.4.2 Testausgaben

8-Bit-VM (ST7, 8 MHz, neue Speicherverwaltung)

```
Jcontrol Benchmarks
Test,Real,User,BCs,OPs,BCs/s,OPs/s,cy/BC,cy/OP
BenchVM(base),2212,2157,,,,,,
BenchVMarithm,4000,3997,14,6,35026,15011,228,533
BenchVMarray,1626,1616,4,1,24752,6188,323,1293
BenchVMcond,1203,1196,4,1,33445,8361,239,957
BenchVMsvar,2843,2835,4,2,14109,7055,567,1134
BenchVMivar,3499,3488,6,2,17202,5734,465,1395
BenchVMsmet,3368,3356,3,1,8939,2980,895,2685
BenchVMimet,4304,4293,4,1,9317,2329,859,3434
BenchVMobj,15379,13473,6,3,4453,2227,1796,3593
BenchVMobj1,6771,5229,1,1,1912,1912,4183,4183
```

JCVM32 (Colibree, Coldfire, 54 MHz)

```
Jcontrol Benchmarks (JCVM32, 67500 loops)
Test,Real,BCs,OPs,BCs/s,OPs/s,cy/BC,cy/OP
BenchVM,2420,0,0,0,0,,
BenchVMarithm,5120,14,6,184570,79101,292,682
BenchVMcond,650,4,1,415384,103846,130,520
BenchVMarray,1090,4,1,247706,61926,218,872
BenchVMsvar,1780,4,2,151685,75842,356,712
BenchVMivar,2700,6,2,150000,50000,360,1080
BenchVMsmet,3620,3,1,55939,18646,965,2896
BenchVMimet,3710,3,1,54582,18194,989,2968
BenchVMobj,278760,6,3,1452,726,37190,74380
BenchVMobj1,291560,1,1,231,231,233766,233766
```

KVM (Pentium M, 1,6 GHz)

```
Jcontrol Benchmarks (J2ME KVM, 20000000 loops)
Test,Real,BCs,OPs,BCs/s,OPs/s,cy/BC,cy/OP
BenchVM,1001,0,0,0,0,,
BenchVMarithm,1452,14,6,192837465,82644628,8,19
BenchVMcond,432,4,1,185185185,46296296,8,34
```

```
BenchVMarray,330,4,1,242424242,60606060,6,26
BenchVMsvar,1112,4,2,71942446,35971223,22,44
BenchVMivar,2585,6,2,46421663,15473887,34,103
BenchVMsmet,1022,3,1,58708414,19569471,27,81
BenchVMimet,1552,3,1,38659793,12886597,41,124
BenchVMobj,4377,6,3,27416038,13708019,58,116
BenchVMobj1,1283,1,1,15588464,15588464,102,102
```

Hotspot (Pentium M, 1,6 GHz, -Xint)

```
Jcontrol Benchmarks (1.5.0_02-b09, x86, Windows XP, 20000000 loops)
Test,Real,BCs,OPs,BCs/s,OPs/s,cy/BC,cy/OP
BenchVM,501,0,0,0,0,,
BenchVMarithm,693,14,6,404040404,173160173,3,9
BenchVMcond,402,4,1,199004975,49751243,8,32
BenchVMarray,256,4,1,312500000,78125000,5,20
BenchVMsvar,737,4,2,108548168,54274084,14,29
BenchVMivar,789,6,2,152091254,50697084,10,31
BenchVMsmet,1431,3,1,41928721,13976240,38,114
BenchVMimet,718,3,1,83565459,27855153,19,57
BenchVMobj,2086,6,3,57526366,28763183,27,55
BenchVMobj1,624,1,1,32051282,32051282,49,49
```

E.5 Messungen am Scheduler

E.5.1 Testprogramme

E.5.1.1 Bibliotheksfunktion: TimerStatistics

Anmerkung Die Klasse `TimerStatistics` liest den statistischen Datensatz der VM aus. Zur Erstellung dieses Datensatzes ist eine Erweiterung der VM-Konfiguration (Patch) erforderlich, diese kann mit dem Schalter

```
oGbl_TimerMeasure equ   USE             ; [undef|USE] führe Messungen des Zeitverhaltens durch
```

in der Datei `Globals.inc` des VM-Kerns aktiviert werden. Da derselbe Speicherbereich für die Messdaten verwendet wird, funktionieren diese Messungen nicht zusammen mit denen der CPU-Zeit (siehe Anhang E.3).

TimerStatistics.java

```
/*
 * (c) Copyright 2004 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
 * All Rights Reserved.
 */

/**
 * TimerStatistics
 * is the user interface for accessing the vm-internal timer statistics counter (if configured).
 * @author boehme
 */
public class TimerStatistics {
    /**
     * Reads the statistics counter values and resets the counter to 0.
     * @return Snapshot of the current timer statistics counter values
     * @see TimerStatistics.asm for native implementation
```

```
     */
    public static native byte[] getValues();
}
```

TimerStatistics.asm

```
;#############################################################################

;; natbib:         TimerStatistics

;#############################################################################

;; header:

;; include:

;*****************************************************************************
;; native:         public static native byte[] getValues();
    dc.w            1, 0                ; Stack, Locals
    dc.w            0, l<id>end - l<id>start
;   input:
;   output:         stack               result in byte[]
;*****************************************************************************
l<id>start:
<code>
                dc.b    $10,$09 ; bipush
                dc.b    $bc,$08 ; newarray
                mBc_CallNative
                move    ,pHp_Temp,y,pFr_opStackH
                move    ,pHp_Temp+1,y,pFr_opStackL   ; TOP
                mBc_GetArraybytes pHp_Temp,pHp_Temp  ; Ziel
                move.w  ,pHp_Temp2,#,$143            ; Quelle
                move.w  ,pHp_Temp3,#,9               ; Anz.
                call    cHp_MemCopy                  ; kopieren
                move.w  ,pHp_Temp2,#,$140            ; Adresse
                move.w  ,pHp_Temp,#,12               ; Anz.
                call    cHp_DirectClear              ; initialisieren
                mBc_LeaveNative
                dc.b    $b0 ; areturn
l<id>end:

```

E.5.1.2 Testprogramm: JitterAnalysis

JitterAnalysis.java

```
/*
 * (c) Copyright 2004 Abt. Entwurf integrierter Schaltungen, TU Braunschweig.
 * All Rights Reserved.
 */

import java.io.IOException;

import jcontrol.comm.ConsoleOutputStream;
import jcontrol.comm.RS232;
import jcontrol.io.Display;
import jcontrol.lang.Math;
import jcontrol.lang.ThreadExt;
import jcontrol.system.Management;

/**
 * JitterAnalysis performs some measurements of thread dispatch jitter using various parameters.
```

```
 * @author boehme
 */
public class JitterAnalysis implements Runnable {

    static ConsoleOutputStream con;
    static Display d=null;

    /**
     * Program entry point. Invokes varoius working threads and performs the measurements.
     * @param args not used
     */
    public static void main(String[] args) {
        try {
            con=new RS232();
            con.println("JControl Scheduler Jitter Analysis");
            con.println("test, count, sum(x), sum(x^2)");

            int[] sizes=new int[] {8, 16, 32, 64, 128, 256};

            doProcess(2,8); // warm up
            con.println("  testing RAM usage...");
            for(int i=0; i<sizes.length; i++) {
                doProcess(2,sizes[i]);
            }

            con.println("  testing Thread usage (running)...");
            for(int i=1; i<9; i++) {
                doProcess(i,16);
            }

            d=new Display();
            con.println("  testing graphics routines...");
            String[] names=new String[] {"Dots", "Lines", "Boxes"};
            for(int i=1; i<4; i++) {
                JitterAnalysis ja=new JitterAnalysis(i);
                con.print(names[i-1].concat(", "));
                Management.gc();
                sleep(1000);                 // tune
                TimerStatistics.getValues(); // reset
                sleep(3000);                 // measure
                byte[] stat=TimerStatistics.getValues();
                ja.stop();
                printStatistics(stat);
            }
            d=null;

            sync=new Object();

            con.println("  testing RAM usage (wait/notify)...");
            for(int i=0; i<sizes.length; i++) {
                doProcess(2,sizes[i]);
            }

            con.println("  testing Thread usage (wait/notify)...");
            for(int i=1; i<9; i++) {
                doProcess(i,16);
            }

            sync=null;

            con.println("  testing Thread usage (sleeping)...");
            for(int i=1; i<9; i++) {
                doProcess(i,0);
            }
```

```
            Management.halt();
        } catch(IOException e) {}
    }

    /**
     * Comnposes the one measurement process.
     * @param numThreads
     * @param blockSize
     * @throws IOException
     */
    private static void doProcess(int numThreads, int blockSize) throws IOException {
        JitterAnalysis[] threads=new JitterAnalysis[numThreads];
            for(int j=0; j<numThreads; j++) {
                threads[j]=new JitterAnalysis(blockSize);
            }
            con.print(String.valueOf(numThreads).concat(" Threads (")
                    .concat(String.valueOf(blockSize)).concat(" bytes alloc), "));
            Management.gc();
            sleep(1000);                // tune
            TimerStatistics.getValues(); // reset
            sleep(3000);                // measure
            byte[] stat=TimerStatistics.getValues();
            for(int j=0; j<numThreads; j++) {
                threads[j].stop();
            }
            printStatistics(stat);
    }

    /**
     * Just sleeps for a specified time.
     * @param millis milliseconds to sleep
     */
    private static void sleep(int millis) {
        try {
            ThreadExt.sleep(millis);
        } catch(InterruptedException e) {}
    }

    /** For byte[] to hex conversion. */
    private static final char[] hex= {'0', '1', '2', '3', '4', '5', '6', '7',
                                      '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'};

    /**
     * Reads VM timer statistics values and prints the results in readable format.
     * @param result of {@link TimerStatistics.getValues();}
     */
    public static void printStatistics(byte[] values) throws IOException {
        con.print("0x");
        for(int i=0; i<2; i++) {
            con.write(hex[(values[i]>>4)&15]);
            con.write(hex[values[i]&15]);
        }
        con.print(", 0x");
        for(int i=2; i<5; i++) {
            con.write(hex[(values[i]>>4)&15]);
            con.write(hex[values[i]&15]);
        }
        con.print(", 0x");
        for(int i=5; i<9; i++) {
            con.write(hex[(values[i]>>4)&15]);
            con.write(hex[values[i]&15]);
        }
        con.println();
    }
```

```
    private int parameter;

    private int running=1;

    private static Object sync=null;

    public JitterAnalysis() {
        this(16);
    }

    public JitterAnalysis(int blocksize) {
        this.parameter=blocksize;
        Thread t=new Thread(this);
        t.setPriority(4);
        t.start();
    }

    /**
     * Main working method. Used to stress the VM in various ways.
     */
    public void run() {
        if(d!=null) {
            switch(parameter) {
            case 1:
                while(running==1) {
                    int x1=Math.rnd(128);
                    int y1=Math.rnd(64);
                    d.setPixel(x1, y1);
                }
                break;
            case 2:
                while(running==1) {
                    int x1=Math.rnd(128);
                    int y1=Math.rnd(64);
                    int x2=Math.rnd(128);
                    int y2=Math.rnd(64);
                    d.drawLine(x1, y1, x2, y2);
                }
                break;
            case 3:
                d.setDrawMode(Display.XOR);
                while(running==1) {
                    int x1=Math.rnd(128);
                    int y1=Math.rnd(64);
                    int x2=Math.rnd(128);
                    int y2=Math.rnd(64);
                    d.fillRect(x1, y1, x2, y2);
                }
                break;
            }
        } else if(parameter==0) {
            while(running==1) {
                try {
                    ThreadExt.sleep(100);
                } catch(InterruptedException e) {}
            }
        } else if(sync!=null){
            synchronized(sync){
                while(running==1) {
                    byte[] buf=new byte[parameter];
                    byte[] buf2=buf;
                    sync.notifyAll();
                    try {
                        sync.wait();
                    } catch(InterruptedException e) {}
```

```
                }
            }
        } else {
            while(running==1) {
                byte[] buf=new byte[parameter];
                byte[] buf2=buf;
            }
        }
        running=0;
    }

    /**
     * Just stops the working thread.
     */
    public void stop() {
        running=2;
        while(running!=0)
            if(sync!=null) synchronized(sync) {
                sync.notifyAll();
            }
            Thread.yield();
    }
}
```

E.5.2 Testausgabe

JitterAnalysis.csv

```
JControl Scheduler Jitter Analysis
test, count, sum(x), sum(x^2)
2 Threads (8 bytes alloc), 102, 53494, 29566134
testing RAM usage...
2 Threads (8 bytes alloc), 102, 56029, 39013249
2 Threads (16 bytes alloc), 115, 55085, 26758781
2 Threads (32 bytes alloc), 156, 75275, 37556023
2 Threads (64 bytes alloc), 199, 95103, 46192285
2 Threads (128 bytes alloc), 254, 124569, 73816541
2 Threads (256 bytes alloc), 298, 142240, 68856024
testing Thread usage (running)...
1 Threads (16 bytes alloc), 79, 31659, 12825875
2 Threads (16 bytes alloc), 113, 54019, 26196779
3 Threads (16 bytes alloc), 149, 83323, 47644041
4 Threads (16 bytes alloc), 185, 117229, 75436843
5 Threads (16 bytes alloc), 221, 157429, 113895147
6 Threads (16 bytes alloc), 249, 196602, 157583886
7 Threads (16 bytes alloc), 295, 256572, 226440348
8 Threads (16 bytes alloc), 322, 305331, 293651685
testing graphics routines...
Dots, 50, 23566, 11819322
Lines, 49, 55169, 118427661
Boxes, 46, 233080, 2218726172
testing RAM usage (wait/notify)...
2 Threads (8 bytes alloc), 875, 435057, 218306993
2 Threads (16 bytes alloc), 877, 436121, 219210207
2 Threads (32 bytes alloc), 856, 422972, 210951616
2 Threads (64 bytes alloc), 842, 416754, 208470604
2 Threads (128 bytes alloc), 816, 404354, 202698930
2 Threads (256 bytes alloc), 714, 351534, 175006158
testing Thread usage (wait/notify)...
1 Threads (16 bytes alloc), 3, 983, 324155
2 Threads (16 bytes alloc), 877, 436387, 219564105
3 Threads (16 bytes alloc), 1196, 690943, 403227631
```

```
4 Threads (16 bytes alloc), 1453, 959986, 641286816
5 Threads (16 bytes alloc), 1589, 1179547, 885821673
6 Threads (16 bytes alloc), 1682, 1386767, 1156979105
7 Threads (16 bytes alloc), 1696, 1536824, 1409357912
8 Threads (16 bytes alloc), 1741, 1722168, 1725172694
testing Thread usage (sleeping)...
1 Threads (0 bytes alloc), 32, 11568, 4194544
2 Threads (0 bytes alloc), 65, 25678, 10427308
3 Threads (0 bytes alloc), 109, 51491, 25244293
4 Threads (0 bytes alloc), 154, 84788, 48566762
5 Threads (0 bytes alloc), 223, 143777, 96613855
6 Threads (0 bytes alloc), 235, 169004, 124867008
7 Threads (0 bytes alloc), 260, 198839, 157076979
8 Threads (0 bytes alloc), 299, 242063, 206207901
```

E.5.3 Messergebnisse

Anmerkung Die Test-Hard- und Software lieferten nur die drei Messwerte *Anzahl* sowie *Summe(x)* und *Summe(x^2)* der Ereignis-Verzögerung, wobei *Anzahl* die Ereignisse bezeichnet, die jeweis im Testzeitraum von 3 Sekunden stattfanden (dies hängt nicht direkt mit dem Rechendurchsatz zusammen). Die übrigen Werte wurden gemäß folgender Anweisungen berechnet:

$$\begin{aligned} \text{Korrekturfaktor:} \quad k &= 48\,\mu s \\ \text{Erwartungswert:} \quad E &= \frac{\sum x}{n} - k \\ \text{Standard-Abweichung:} \quad \sigma &= \sqrt{\frac{\sum x^2}{n} - (E+k)^2} \end{aligned}$$

1. Durchlauf: Running

Test	Anzahl	Sum(x)[μs]	Sum(x^2)[μs^2]	E.-Wert[μs]	Std-A.[μs]
2 Thr, 8 B	102	56029	39013249	501	284
2 Thr, 16 B	115	55085	26758781	431	57
2 Thr, 32 B	156	75275	37556023	435	89
2 Thr, 64 B	199	95103	46192285	430	61
2 Thr, 128 B	254	124569	73816541	442	224
2 Thr, 256 B	298	142240	68856024	429	57
1 Thr, 16 B	79	31659	128258 75	353	42
2 Thr, 16 B	113	54019	26196779	430	57
3 Thr, 16 B	149	83323	47644041	511	84
4 Thr, 16 B	185	117229	75436843	586	79
5 Thr, 16 B	221	157429	113895147	664	89
6 Thr, 16 B	249	196602	157583886	742	97

Test	**Anzahl**	**Sum(x)**[μs]	**Sum(x²)**[μs²]	**E.-Wert**[μs]	**Std-A.**[μs]
7 Thr, 16 B	295	256572	226440348	822	106
8 Thr, 16 B	322	305331	293651685	900	113
Dots	50	23566	11819322	423	119
Lines	49	55169	118427661	1078	1072
Boxes	46	233080	2218726172	5019	4750

2. Durchlauf: Wait/Notify

Test	**Anzahl**	**Sum(x)**[μs]	**Sum(x²)**[μs²]	**E.-Wert**[μs]	**Std-A.**[μs]
2 Thr, 8 B	875	435057	218306993	449	48
2 Thr, 16 B	877	436121	219210207	449	52
2 Thr, 32 B	856	422972	210951616	446	48
2 Thr, 64 B	842	416754	208470604	447	51
2 Thr, 128 B	816	404354	202698930	448	53
2 Thr, 256 B	714	351534	175006158	444	52
1 Thr, 16 B	3	983	324155		
2 Thr, 16 B	877	436387	219564105	450	53
3 Thr, 16 B	1196	690943	403227631	530	58
4 Thr, 16 B	1453	959986	641286816	613	70
5 Thr, 16 B	1589	1179547	885821673	694	80
6 Thr, 16 B	1682	1386767	1156979105	776	90
7 Thr, 16 B	1696	1536824	1409357912	858	99
8 Thr, 16 B	1741	1722168	1725172694	941	111

3. Durchlauf: Sleeping

Test	**Anzahl**	**Sum(x)**[μs]	**Sum(x²)**[μs²]	**E.-Wert**[μs]	**Std-A.**[μs]
1 Thr, 0 B	32	11568	4194544	314	20
2 Thr, 0 B	65	25678	10427308	347	66
3 Thr, 0 B	109	51491	25244293	424	92
4 Thr, 0 B	154	84788	48566762	503	111
5 Thr, 0 B	223	143777	96613855	597	133
6 Thr, 0 B	235	169004	124867008	671	119
7 Thr, 0 B	260	198839	157076979	717	139
8 Thr, 0 B	299	242063	206207901	762	185

Lebenslauf

Name:	Helge Böhme
Geboren am:	21. 5. 1972 in Northeim
Eltern:	Jürgen Böhme, Verwaltungsbeamter (a. D.) Marianne Gorf, geb. Kickstein, Grundschullehrerin
Familienstand:	ledig

Schulbildung

1978 bis 1982	Martin-Luther-Grundschule in Northeim
1982 bis 1985	Thomas-Mann-Orientierungsstufe in Northeim
1985 bis 1989	Gymnasium Corvinianum in Northeim
1989 bis 1992	Fachgymnasium Technik an den BBSII in Göttingen
26. 5. 1992	**Abitur**

Studium

1992 bis 1999	Elektrotechnik (Diplom) an der TU Braunschweig
14. 7. 1999	**Dipl.-Ing. Elektrotechnik**
2006	Promotionsgesuch zum Dr.-Ing.

Praktika

1989 bis 1992	acht Wochen Fachpraxis an den BBSII in Göttingen
1995	vier Wochen Feinmechanisches Praktikum am MPI in Lindau
1996	acht Wochen Fertigungs-, Test- und Entwicklungspraktikum bei der Firma **aticon** in Braunschweig
1997	sechs Wochen Test- und Entwicklungspraktikum bei der IAM FuE GmbH in Braunschweig

Berufliche Laufbahn

1996 bis 1998	freier Mitarbeiter bei der Firma **aticon** in Braunschweig
1999 bis 2005	wissenschaftlicher Mitarbeiter an der Abteilung Entwurf integrierter Schaltungen (E.I.S.) der TU Braunschweig

Inhalt

Diese Dissertation zeigt, wie Java auf besonders preiswerten 8-Bit-Mikrocontrollern ausgeführt werden kann. Das eröffnet für Java die Welt der Messung, Steuerung und Regelung und verknüpft dies mit Benutzerinteraktion und Kommunikation. Java kann mit geringem Programmieraufwand dazu beitragen, beispielsweise einen Haushalt zu steuern und zu überwachen.

Vorgestellt werden Techniken zur speichereffizienten Implementierung einer JavaVM und von Java-Anwendungen. Die Kombination von Java-Programmen mit zielsystemabhängigen Code ermöglicht die Ansteuerung von Peripheriekomponenten. Ferner werden spezielle Zeitsteuerungen integriert, die Java auch für zeitkritische Anwendungen geeignet machen. Zusammen mit einer Software-Umgebung auf dem Entwicklungssystem entstand ein einsatzfähiges Java-System für einen Mikrocontroller.